BBN-LISP

TENEX Reference Manual
July 1971

W. Teitelman
D.G. Bobrow
A.K. Hartley
D.L. Murphy

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, A.K. Hartley, D.L. Murphy, and W. Teitelman. Most of the extended interactive features are the work of Warren Teitelman and the existence of this extensive reference manual is due primarily to his perseverence and effort. TRANSOR, the special arithmetic functions, and a number of other utility functions were written by J.W. Goodwin. The authors are grateful to Karen O'Sullivan for her hard work and patience through endless revisions of the text, and to Kathryn Paiva for her valuable assistance.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

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 DEC PDP-10 under the BBN TENEX time sharing system. BBN-LISP is designed to provide the user access to the large virtual memory allowed by TENEX, with a relatively small penalty in speed (using special paging techniques described in Bobrow and Murphy, 1967). Additional data types have been added, including strings and hash association tables (hash links). 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. Machine code can be intermixed with LISP expressions via the assemble directive of the compiler. 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.

BBN LISP provides three levels of computation: a LISP interpreter, a compatible function compiler and a block compiler, which allows a group of functions to be compiled as a unit, suppressing internal names. Each successive level provides greater speed at a cost of debugging ease.

To aid in converting to BBN-LISP programs written in other LISP dialects, e.g., LISP 1.5, Stanford LISP, we have implemented TRANSOR, a subsystem which accepts transformations (or can operate from previously defined transformations), and applies these transformations to source programs written in another LISP dialect, producing object programs which will run on BBN LISP. In addition, TRANSOR alerts the programmer to problem areas that (may) need further attention. TRANSOR was used extensively in converting from 940 LISP to BBN-LISP on the PDP-10. A set of transformations is available for converting from Stanford LISP and LISP 1.5 to BBN LISP.

In addition to the sub-systems described in this manual, a complete format directed list processing sub-system (FLIP, Teitelman, 1967) is available for use within BBN LISP.

Although we have tried to be as clear and complete as possible, this document is not designed to be an introduction to LISP. Therefore, some parts may 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, 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 Programmer's 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 LISP

Using the LISP Manual - Format, Notation, and Conventions

The LISP manual is divided into separate more or less independent sections. Each section is paginated independently, i.e., Section 4 contains pages 4.1 to 4.4. This is to facilitate issuing updates of sections. Each section begins with a list of key words, functions, and variables contained in the section, and a rough approximation of their location, i.e., a mini-table of contents. In addition, there will be a complete index of functions and variables for the entire manual, plus several appendices and a table of contents.

Throughout the manual, terminology and conventions will be offset from the text and typed in italics, frequently at the beginning of a section. For example, one such notational convention is:

The names of functions and variables are written in lower case and underlined when they appear in the text. Meta-LISP notation is used for describing forms.

Examples: member[x;y] is equivalent to (MEMBER X Y), member[car[x];FOO] is equivalent to (MEMBER (CAR X) (QUOTE FOO)). Note that in meta-LISP notation lower case variables are evaluated, upper case quoted.

. *notation is used to distinguish between* *<u>cons</u>* *and* *<u>list</u>*.

e.g., if x=(A B C), (FOO x) is (FOO (A B C)), whereas (FOO . x) is (FOO A B C). Note that this convention is in fact followed by the read program, i.e., (FOO . (A B C)) and (FOO A B C) read in as equal structures.

Other important conventions are:

TRUE in BBN-LISP means not NIL.

The purpose of this is to allow a single function to be used both for the computation of some quantity, and as a test for a condition. For example, the value of member[x;y] is either NIL, or the tail of y beginning with x. Similarly, the value of or is the value of its first TRUE, i.e., non-NIL, expression, and the value of and is either NIL, or the value of its last expression.

Although most lists terminate in NIL, the occasional list that ends in an atom, e.g., (A B . C) or worse, a number or string, could cause bizarre effects. Accordingly, we have made the following implementation decision:

All functions that iterate through a list, e.g., member, length, mapc, etc. terminate by an nlistp check, rather than the conventional null-check, as a safety precaution against encountering data types which might cause infinite cdr loops, e.g., strings, numbers, arrays.

Thus, member[x;(A B . C)]=member[x;(A B)]
reverse[(A B . C)]=reverse[(A B)]
append[(A B . C);y]=append[(A B);y]

For users with an application requiring extreme efficiency,* we have provided fast versions of member, last, nth, assoc, and length which compile open and terminate on NIL checks, and therefore may cause infinite cdr loops if given poorly formed arguments.

*A NIL check can be executed in only one instruction, an nlistp requires about 12, although both generate only one word of code.

Most functions that set system parameters, e.g., printlevel, linelength, radix, etc., return as their value the old setting. If given NIL as an argument, they return the current value without changing it.

All SUBRS, i.e., hand coded functions, such as read, print, eval, cons, etc., have 'argument names' (U V W) as described under arglist, section 8. However, for tutorial purposes, more suggestive names are used in the descriptions of these functions in the text.

Most functions whose names end in p are predicates, e.g., numberp, tailp, exprp; most functions whose names end in q are nlambda's, i.e., do not require quoting their arguments, e.g., setq, defineq, nlsetq.

"x is equal to y" means equal[x;y] is true, as opposed to "x is eq to y" meaning eq[x;y] is true, i.e., x and y are the same identical LISP pointer.

When new literal atoms are created (by the read program, pack, or mkatom), they are provided with a function definition cell initialized to NIL (Section 8), a value cell initialized to the atom NOBIND (Section 16), and a property list initialized to NIL (Section 7). The function definition cell is accessed by the functions getd and putd described in Section 8. The value cell of an atom is car of the atom, and its property list is cdr of the atom. In particular, car of NIL and cdr of NIL are always NIL, and the system will resist attempts to change them (p. 5.6, p. 5.8).

The term list refers to any structure created by one or more conses, i.e. it does not have to end in NIL. For example, (A . B) is a list. The function listp, Section 5, is used to test for lists. Note that not being a list does not necessarily imply an atom, e.g., strings and arrays are not lists, nor are they atoms. See Section 10.

Using the LISP System on TENEX - An Overview

Call LISP by typing LISP followed by a carriage return. LISP will type an identifying message, the date, and a greeting, followed by a '←'. This prompt character indicates that the user is "talking to" the top level LISP executive, evalqt (Section 22), just as '@' indicates the user is talking to TENEX. evalqt calls lispx which accepts inputs in either eval, or apply format: if just one expression is typed on a line, it is evaluated; if two expressions are typed, the first is apply-ed to the second. In both cases, the value is typed, followed by ← indicating LISP is ready for another input.

LISP is normally exited via the function LOGOUT, i.e., the user types LOGOUT(). However, typing control-C at any point in the computation returns control immediately to TENEX. The user can then *continue* his program with no ill effects with the TENEX CONTINUE command, even if he interrupted it during a garbage collection. Or he can *reenter* his program at evalqt with the TENEX REENTER command. *The latter is DEFINITELY not advisable if the Control-C was typed during a garbage collection.* Typing control-D at any point during a computation will return control to evalqt. If typed during a garbage collection, the garbage collection will first be completed, and *then* control will be returned to LISP's top level, otherwise, control returns immediately.

When typing to the LISP read program, typing a control-Q will cause LISP to print '##' and clear the input buffer, i.e., erase the entire line up to the last carriage return. Typing control-A erases the last character typed in, echoing a \ and the erased character. Control-A will not back up beyond the last carriage return. Control-O can be used to *immediately* clear the output buffer, and rubout to *immediately* clear the input buffer.* In addition, typing control-U (in most cases) will cause the LISP editor (Section 9) to be called on the expression being read, when the read is completed. Appendix 3 contains a list of all control characters, and a reference to that part of the manual where they are described.

Since the LISP read program is normally line buffered to make possible the action of control-Q,** the user must type a carriage return before any characters are delivered to the function requesting input, e.g.,

```
←E T↵
T
```

However, the read program *automatically* supplies (and prints) this carriage return when a matching right parenthesis is typed, making it unnecessary for the user to do so, e.g.,

```
←CONS(A B)
(A . B)
```

The LISP read program treats square brackets as 'super-parentheses': a right square bracket automatically supplies enough right parentheses to match back to the last left square bracket (in the expression being read), or if none has appeared, to match the first left parentheses,

e.g., (A (B (C]=(A (B (C))),
      (A [B (C (D] E)=(A (B (C (D))) E).

---

*The action of control-Q takes place when it is *read*. If the user has 'typed ahead' several inputs, control-Q will only affect at most the last line of input. Rubout however will clear the input buffer when it is *typed*, i.e., even during a garbage collection.

**Except following control[T], see Section 14.

% is the universal escape character for read. Thus to input an atom containing a syntactic delimiter, precede it by %, e.g. AB%(C or %%. See Section 14 for more details.

SECTION III

DATA TYPES, STORAGE ALLOCATION, AND GARBAGE COLLECTION

LISP operates in an 18-bit address space. This address space is divided into 512 word pages with a limit of 512 pages, or 262,144 words, but only that portion of address space currently in use actually exists on any storage medium. LISP itself and all data storage are contained within this address space. A pointer to a data element such as a number, atom, etc., is simply the address of the data element in this 18-bit address space.

Data Types

The data types of BBN-LISP are lists, atoms, pnames, arrays, large and small integers, floating point numbers, string characters and string pointers. Compiled code and hash arrays are currently included with arrays.

In the descriptions of the various data types given below, for each data type, first the input syntax and output format are described, that is, what input sequence will cause the LISP read program to construct an element of that type, and how the LISP print program will print such an element. Next, those functions that construct elements of that data type are given. Note that some data types cannot be input, they can only be constructed, e.g. arrays. Finally, the format in which an element of that data type is stored in memory is described.

Literal Atoms

A literal atom is input as any string of non-delimiting characters that cannot be interpreted as a number. The syntactic characters that delimit atoms are space, end-of-line,[†] line feed, % ( ) "] and [. However, these characters may be included in atoms by preceding them with the escape character %.

Literal atoms are printed by print and prin2 as a sequence of characters with %'s inserted before all delimiting characters (so that the atom will read back in properly). Literal atoms are printed by prinl as a sequence of characters without these extra %'s. For example, the atom consisting of the five characters A, B, C, (, and D will be printed as ABC%(D by print and ABC(D by prinl. The extra %'s are an artifact of the print program; they are not stored in the atom's pname.

Literal atoms can be constructed by pack, mkatom, and gensym, (which uses mkatom).

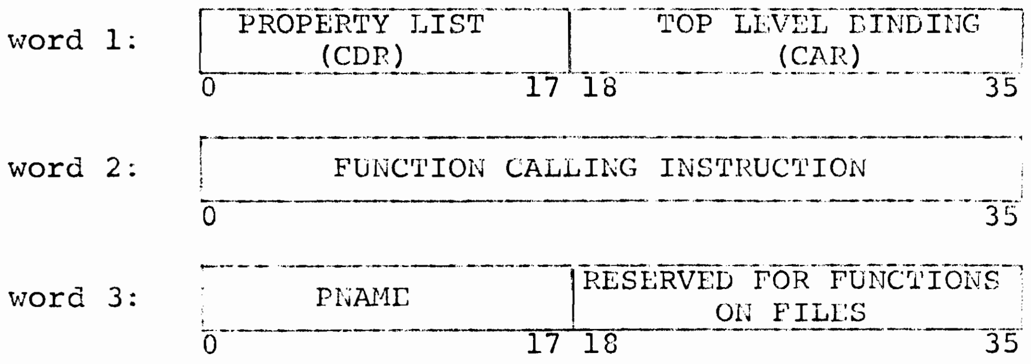
Literal atoms are unique. In other words, if two literal atoms have the same pname, i.e. print the same, they will *always* be the same identical atom, that is, they will always have the same address in memory, or equivalently, they will always be eq.* Thus if pack or mkatom is given a list of characters corresponding to a literal atom that already exists, they return a pointer to that atom, and do *not* make a new atom. Similarly, if the read program is given as input of a sequence of characters for which an atom already exists, it returns a pointer to that atom.

---

†An end-of-line character is transmitted by TENEX when it sees a carriage return.

*Note that this is *not* true for strings, large integers, floating point numbers, and lists, i.e. they all can print the same without being eq.

A literal atom is a 3 PDP-10) word datum containing:



Car of a literal atom, i.e. the right half of word 1, contains its top level binding, initially the atom NOBIND. *Cdr* of the atom is a pointer to its property list, initially NIL.

Word 2, the function cell, is a full PDP-10 word, containing an instruction to be executed for calling the function associated with that atom, if any. The left half differs for different function types (i.e., EXPR, SUBR, or compiled code); the right half is a pointer to the function definition.[†]

The pname cell, the left half of the third word, contains a pointer to the pname of the atom. The remaining half word is reserved for an extension of LISP to permit storing function definitions on files.

[†]This use of a full word saves some time in function calls from compiled code in that we do not need to look up the type of the function definition at call time.

Pnames

The pnames of atoms,[†] pointed to in the third word of the atom, comprise another data type with storage assigned as it is needed. This data type only occurs as a component of an atom or a string. It does not appear, for example, as an element of a list.

Pnames have no input syntax or output format as they cannot be directly referenced by user programs.

A pname is a sequence of 7 bit characters packed 5 to a word, beginning at a word boundary. The first character of a pname contains its length; thus the maximum length of a pname is 126 characters.

[†]All BBN-LISP pointers have pnames, since we define a pname simply to be how that pointer is printed. However, only literal atoms and strings have their pnames explicitly stored. Thus, the use of the term pname in a discussion of data types or storage allocation means pnames of atoms or strings, and refers to a sequence of characters stored in a certain part of LISP's memory.

Numerical Atoms

Numerical atoms, or simply numbers, do not have property lists, value cells, function definition cells, or explicit pnames. There are currently two types of numbers in BBN-LISP: integers, and floating point numbers.

Integers

The input syntax for an integer is an optional sign (+ or -) followed by a sequence of digits, followed by an optional Q.* If the Q is present, the digits are interpreted in octal, otherwise in decimal, e.g. 77Q and 63 both correspond to the same integers, and in fact are indistinguishable internally since no record is kept of how the integers were created.

The setting of radix, p. 14.18, determines how integers are printed: signed or unsigned, octal or decimal.

Integers are created by pack and mkatom when given a sequence of characters observing the above syntax, e.g. (PACK (LIST 1 2 (QUOTE Q))) = 10. Integers are also created as a result of arithmetic operations, as described in Chapter 13.

---

*and terminated by a delimiting character. Note that some data types are self-delimiting, e.g. lists.

An integer is stored in one PDP-10 word; thus its magnitude must be less that 2^{35}.[†] To avoid having to store (and hence garbage collect) the values of small integers, a few pages of address space, overlapping the LISP machine language code, are reserved for their representation. The small number pointer *itself*, minus a constant, is the value of the number. Currently the range of 'small' integers is -1536 thru +1535. The predicate smallp is used to test whether an integer is 'small'.

While small integers have a unique representation, large integers do not. In other words, two large integers may have the same value, but not the same address in memory, and therefore not be eq. For this reason the function eqp (or equal) should be used to test equality of large integers.

[†] If the sequence of digits used to create the integer is too large, the high order portion is discarded. (The handling of overflow as a result of arithmetic operations is discussed in Section 13.)

Floating Point Numbers

A floating point number is input as a signed integer, followed by a decimal point, followed by another sequence of digits called the fraction, followed by an exponent (represented by E followed by a signed integer).* Both signs are optional, and either the fraction following the decimal point, or the integer preceding the decimal point may be omitted. One or the other of the decimal point or exponent may also be omitted, but at least one of them must be present to distinguish a floating point number from an integer. For example, the following will be recognized as floating point numbers:

```
5.      5.00      5.01      .3      5E2      5.1E2
                  5E-3      -5.2E+6
```

Floating point numbers are printed using the facilities provided by TENEX. LISP calls the floating point number to string conversion routines† using the format control specified by the function <u>fltfmt</u>. <u>fltfmt</u> is initialized to 0, or free format. For example, the above floating point numbers would be printed in free format as:

```
5.0      5.0      5.01      .3      500.0      510.0
                 .005      -5.2E6
```

Floating point numbers are also created by <u>pack</u> and <u>mkatom</u>, and as a result of arithmetic operations as described in Chapter 13.

A floating point number is stored in one PDP-10 word in standard PDP-10 format. The range is $\pm 2.94E-39$ thru $\pm 1.69E38$ (or $1*2^{-128}$ thru $1*2^{127}$).

---

* and terminated by a delimiter.

† Additional information concerning these conversions may be obtained from the TENEX JSYS Manual.

Lists

The input syntax for a list is a sequence (at least one)* of LISP data elements, e.g. literal atoms, numbers, other lists, etc. enclosed in parentheses or brackets. A bracket can be used to terminate several lists, e.g. (A (B (C], as described on page 2.5.

If there are two or more elements in a list, the final element can be preceded by a . (delimited on both sides), indicating that cdr of the final node in the list is to be the element immediately following the . , e.g. (A . B) or (A B C . D), otherwise cdr of the last node in a list will be NIL.** Note that the input sequence (A B C . NIL) is thus equivalent to (A B C), and that (A B . (C D)) is thus equivalent to (A B C D). Note however that (A B . C D) will create a list containing the five literal atoms A B . C and D.

Lists are constructed by the primitive functions cons and list.

Lists are printed by printing a left parenthesis, and then printing the first element of the list[†], then printing a space, then printing the second element, etc. until the final node is reached. Lists are considered to terminate when cdr of some node is not a list. If cdr of this terminal node is NIL (the usual case), car of the terminal node is printed followed by a right parenthesis. If cdr of the terminal node is *not* NIL, car of the terminal node is printed, followed by a space, a period, another space, cdr of the terminal node, and then the right parenthesis. Note that a list input as (A B C . NIL)

---

* () is read as the atom NIL.

** Note that in BBN LISP terminology, a list does *not* have to end in NIL, it is simply a structure composed of one or more conses.

† The individual elements of a list are printed using prin2 if the list is being printed by print or prin2, and by prin1 if the list is being printed by prin1.

will print as (A B C), and a list input as (A B . (C D)) will print as (A B C D). Note also that printlevel affects the printing of lists to teletype, as described on page 14.13, and that carriage returns may be inserted where dictated by linelength, as described on page 14.18.

A list is stored as a chain of list *nodes*. A list node is stored in one PDP-10 word, the right half containing car of the list (a pointer to the first element of the list) and the left half containing cdr of the list (a pointer to the next node of the list).

Arrays

An array in LISP is a one dimensional block of contiguous storage of arbitrary length. Arrays do not have input syntax, they can only be created by the function array. Arrays are printed by both print, prin2, and prinl, as # followed by the address of the array pointer (in octal). Array elements can be referenced by the functions elt and eltd, and set by the functions seta and setd, as described in chapter 10.

Arrays are partitioned into four sections: a header, a section containing unboxed numbers, a section containing LISP pointers, and a section containing relocation information. The last three sections can each be of arbitrary length (including 0); the header is two words long and contains the length of the other sections as indicated in the diagram below. The unboxed number region of an array is used to store 36 bit quantities that are not LISP pointers, and therefore not to be chased from during garbage collections, e.g. machine instructions. The relocation information is used when the array contains the definition of a compiled function, and specifies which locations in the *unboxed* region of the array must be changed if the array is moved during a garbage collection.

The format of an array is as follows:

HEADER	WORD Ø	ADDRESS OF RELOCATION INFORMATION	LENGTH
	WORD 1	USED BY GARBAGE COLLECTOR	ADDRESS OF POINTERS
FIRST DATA WORD		NON-POINTERS	
		POINTERS	
		RELOCATION INFORMATION	

The header contains:

word Ø right - length of entire block=ARRAYSIZE+2.
       left - address of relocation information relative to word Ø of block (>Ø if relocation information exists, negative if array is a hash array, Ø if ordinary array).

word 1 right - address of pointers relative to word Ø of block.
       left - used by garbage collector.

Strings

The input syntax for a string is a ", followed by a sequence of any characters except " and %, terminated by a ". " and % may be included in a string by preceding them with the escape character %.

Strings are printed by print and prin2 with initial and final "'s, and %'s inserted where necessary for it to read back in properly. Strings are printed by prin1 without the delimiting "'s and extra %'s.

Strings are created by mkstring, substring, and concat.

Internally a string is stored in two parts; a string pointer and the sequence of characters. The LISP pointer to a string is the address of the string pointer. The string pointer, in turn, contains the character position at which the string characters begin, and the number of characters. String pointers and string characters are two separate data types,[†] and several string pointers may reference the same characters. This method of storing strings permits the creation of a substring by creating a new string pointer, thus avoiding copying of the characters. For more details, see p. 10.10.

String characters are 7 bit bytes packed 5 to a (PDP-10) word. The format of a string pointer is

```
 -----------------------------------------------------------------------
| # OF CHARACTERS  | 5 * ADDRESS OF STRING + CHARACTER                  |
|                  |                 POSITION                           |
 -----------------------------------------------------------------------
0                14 15                                                35
```

The maximum length of a string is 32K (K=1024) characters.

[†] String characters are not directly accessible by user programs.

Storage Allocation and Garbage Collection

In the following discussion, we will speak of a quantity of memory being *assigned* to a particular data type, meaning that the space is reserved for storage of elements of that type. *Allocation* will refer to the process used to obtain from the already assigned storage a particular location for storing one data element.

A small amount of storage is assigned to each data type when LISP is started; additional storage is assigned only during a garbage collection.

The page is the smallest unit of memory that may be assigned for use by a particular data type. For each page of memory there is a one word entry in a type table. The entry contains the data type residing on the page as well as other information about the page. The type of a pointer is determined by examining the appropriate entry in the type table.

Storage is allocated as is needed by the functions which create new data elements, such as cons, pack, mkstring. For example, when a large integer is created by iplus, the integer is stored in the next available location in the space assigned to integers. If there is no available location, a garbage collection is initiated, which may result in more storage being assigned.

The storage allocation and garbage collection methods differ for the various data types. The major distinction is between the types with elements of fixed length and the types with elements of arbitrary length. List *nodes*, atoms, large integers, floating point numbers, and string pointers are fixed length; all occupy 1 word except atoms which use 3 words. Arrays, pnames, and strings are variable length.

Elements of fixed length types are stored so that they do not overlap page boundaries. Thus the pages assigned to a fixed length type need not be adjacent. If more space is needed, any empty page will be used. The method of *allocating* storage for these types employs a free-list of available locations; that is, each available location contains a pointer to the next available location. A new element is stored at the first location on the free-list, and the free-list pointer is updated.*

Elements of variable length data types *are* allowed to overlap page boundaries. Consequently all pages assigned to a particular variable length type must be contiguous. Space for a new element is allocated following the last space used in the assigned block of contiguous storage.

When LISP is first called, a few pages of memory are assigned to each data type. When the allocation routine for a type determines that no more space is available in the assigned storage for that type, a garbage collection is initiated. The garbage collector determines what data is currently in use and reclaims that which is no longer in use. A garbage collection may also be initiated by the user with the function reclaim. (see p. 10.14).

Data in use (also called active data) is any data that can be 'reached' from the currently running program (i.e., variable bindings and functions in execution) or from atoms. To find the active data the garbage collector 'chases' all pointers, beginning with the contents of the push-down lists and the components (i.e., car, cdr, and function definition cell) of all atoms with at least one non-trivial component.

* The allocation routine for list nodes is more complicated. Each page containing list nodes has a separate free list. First a page is chosen (see CONS for details), then the free list for that page is used. Lists are the only data type which operate this way.

When a previously unmarked datum is encountered, it is marked, and all pointers contained in it are chased. Most data types are marked using bit tables; that is tables containing one bit for each datum. Arrays, however, are marked using a half-word in the array header.

When the mark and chase process is completed, unmarked (and therefore unused) space is reclaimed. Elements of fixed length types that are no longer active are reclaimed by adding their locations to the free list for that type. This free list allocation method permits reclaiming space without moving any data, thereby avoiding the time consuming process of updating all pointers to moved data. To reclaim unused space in a block of storage assigned to a *variable* length type, the active elements are compacted toward the beginning of the storage block, and then a scan of all active data that can contain pointers to the moved data is performed to update the pointers.

Whenever a garbage collection of any type is initiated,* unused space for all fixed length types is reclaimed since the additional cost is slight. However, space for a variable length type is reclaimed only when that type initiated the garbage collection.

* The 'type of a garbage collection' or the 'type that initiated a garbage collection' means either the type that ran out of space and called the garbage collector, or the argument to __reclaim__.

If the amount of storage reclaimed for the type that initiated the garbage collection is less than the minimum free storage requirement for that type, the garbage collector will assign enough additional storage to satisfy the minimum free storage requirement. The minimum free storage requirement for each data type may be set with the function minfs, p. 10.15. The garbage collector assigns additional storage to fixed length types by finding empty pages, and adding the appropriate size elements from each page to the free list. Assigning additional storage to a variable length type involves finding empty pages and moving data so that the empty pages are at the end of the block of storage assigned to that type.

In addition to increasing the storage assigned to the type initiating a garbage collection, the garbage collector will attempt to minimize garbage collections by assigning more storage to other *fixed* length types according to the following algorithm.* If the amount of *active* data of a type has increased since the last garbage collection by more than 1/4 of the minfs value for that type, storage is increased (if necessary) to attain the minfs value. If active data has increased by less than 1/4 of the minfs value, available storage is increased to 1/2 minfs. If there has been no increase, no more storage is added. For example, if the minfs setting is 2000 words, the number of active words has increased by 700, and after all unused words have been collected there are 1000 words available, 1024 additional words (two pages) will be assigned to bring the total to 2024 words available. If the number of active words had increased by only 300, and there were 500 words available, 512 additional words would be assigned.

* We may experiment with different algorithms.

Shared LISP

The LISP system initially obtained by the user is shared; that is, all active users of LISP are actually using the same pages of memory. As a user adds to the system, private pages are added to his memory. Similarly, if the user changes anything in the original shared LISP, for example, by advising a system function, a private copy of the changed page is created.

In addition to the swapping time saved by having several users accessing the same memory, the sharing mechanism permits a large saving in garbage collection time, since we do not have to garbage collect any data in the shared system, and thus do not need to chase from any pointers on shared pages during garbage collections.

This reduction in garbage collection time is possible because the shared system usually is not modified very much by the user. If the shared system is changed extensively, the savings in time will vanish, because once a page that was initially shared is made private, every pointer on it must be assumed active, because it may be pointed *to* by something in the shared system. Since every pointer on an initially shared but now private page can also point to *private* data, they must always be chased.

A user may create his own shared system with the function makesys. If several people are using the same system, making the system be shared will result in a savings in swapping time. Similarly, if a system is large and seldom modified, making it be shared will result in a reduction of garbage collection time, and may therefore be worthwhile even if the system is only being used by one user.

SECTION IV

FUNCTION TYPES AND IMPLICIT PROGN

In BBN LISP, each function may independently have:

a. its arguments evaluated or not evaluated;

b. a fixed number of arguments or an indefinite number of arguments;

c. be defined by a LISP expression, by built-in machine code, or by compiled machine code.

Hence there are twelve function types (2 x 2 x 3).

Exprs

Functions defined by LISP expressions are called exprs. Exprs must begin with either LAMBDA or NLAMBDA,* indicating whether the arguments to the function are to be evaluated or not evaluated, respectively. Following the LAMBDA or NLAMBDA in the expr is the 'argument list', which is either

(1) a list of literal atoms or NIL (fixed number of arguments); or

(2) any literal atom other than NIL, (indefinite number of arguments).

---

* Where unambiguous, the term expr is used to refer to either the function, or its definition.

Case (1) corresponds to a function with a *fixed* number of arguments. Each atom in the list is the *name* of an argument for the function defined by this expression. Arguments for the function will be evaluated or not evaluated, as dictated by whether the definition begins with LAMBDA or NLAMBDA, and then paired with these argument names. This process is called "spreading" the arguments, and the function is called a spread-LAMBDA or a spread-NLAMBDA.

Case (2) corresponds to a function with an *indefinite* number of arguments. Such a function is called a nospread function. If its definition begins with NLAMBDA, the atom which constitutes its argument list is bound to the list of arguments to the function (unevaluated). For example, if FOO is defined by (NLAMBDA X --), when (FOO THIS IS A TEST) is evaluated, X will be bound to (THIS IS A TEST).

If a nospread function begins with a LAMBDA, indicating its arguments are to be evaluated, each of its n arguments are evaluated and their values stored on the pushdown list. The atom following the LAMBDA is then bound to the *number* of arguments which have been evaluated. For example, if FOO is defined by (LAMBDA X --) when (FOO A B C) is evaluated, A, B, and C are evaluated and X is bound to 3. A built-in function arg[atm;m] is available for computing the value of the mth argument for the lambda-atom variable atm. arg is described in section 8.

Compiled Functions

Functions defined by expressions can be compiled by the LISP compiler, as described in section 18, "The Compiler and Assembler". Functions 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.

Function Type

The function fntyp[fn] returns the function type of fn. The value of fntyp is one of the following 12 types:

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

The types in the first column are all defined by expressions. 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 have a fixed number of arguments, i.e., are spread functions. Functions in the third and fourth rows have an indefinite number of arguments, as indicated by the suffix *. The prefix F indicates no evaluation of arguments. Thus, for example, a CFEXPR* is a compiled form of a nospread-NLAMBDA.

A standard feature of the BBN LISP system is that no error occurs if a spread 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. In fact, the function itself cannot distinguish between being given NIL as an argument, and not being given that argument, e.g., (FOO) and (FOO NIL) are exactly the same.

Progn

progn is a function of an arbitrary number of arguments. progn evaluates the arguments in order and returns the value of the last, i.e., it is an extension of the function prog2 of LISP 1.5. Both cond and lambda/nlambda expressions have been generalized to permit 'implicit progns' as described below.

Implicit Progn

The conditional expression has been generalized so that each clause may contain n forms (n≥1) which are interpreted as follows:

```
(COND
   (P1 E11 E12 E13)
   (P2 E21 E22)                          [1]
   (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)                               [2]
   (P4 E41)
   (T NIL))
```

Note however that P3 is evaluated only once in [1], while it is evaluated a second time if the expression is written as in [2]. Thus a list in a cond with only a predicate and no following expression causes the value of the predicate itself to be returned. Note also that NIL is returned if all the predicates have value NIL, i.e., the cond 'falls off the end'. No error is generated.

LAMBDA and NLAMBDA expressions also allow 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 entire expression. In this example, the function would always return NIL.

SECTION V

PRIMITIVE FUNCTIONS AND PREDICATES

Contents

Primitive Functions

car[x]

car gives the first element of a list x, or the left element of a dotted pair x. For literal atom, value is top level binding (value) of the atom. For all other nonlists, e.g. strings, arrays, and numbers, the value is undefined, i.e., it is the right 18 bits of x.

cdr[x]

cdr gives the rest 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. The value of cdr is undefined for other nonlists, i.e. it is the left 18 bits of x.

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

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

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

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

* Means car is on page 5.1, rplacd on page 5.2, cond on 5.4, etc.

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 any page with a specified minimum of storage, presently 16 LISP words.

conscount[]	Value is the number of conses since this LISP was started up.
rplacd[x;y]	Places the pointer y in the decrement, i.e. cdr, of the cell pointed to by x. Thus it physically changes the internal list structure of x, as opposed to cons which creates a new list element. The only way to get a circular list is by using rplacd to place a pointer to the beginning of a list in a spot at the end of the list.

The value of rplacd is x. An attempt to rplacd NIL will cause an error (except for rplacd[NIL;NIL]). For x a literal atom, rplacd[x;y] will make y be the property list of x. For all other non-lists, rplacd should be used with care: it will simply store y in the left 18 bits of x.

rplaca[x;y]

similar to rplacd, but replaces the address pointer of x, i.e., car, with y. The value of rplaca is x. An attempt to rplaca NIL will cause an error, (except for rplaca[NIL;NIL]). For x a literal atom, rplaca[x;y] will make y be the top level value for x. For all other non-lists, rplaca should be used with care: it will simply store y in the right 18 bits of x.

Convention: Naming a function by prefixing an existing function name with f usually indicates that the new function is a fast version of the old, i.e., one which has the same definition but compiles open and runs without any 'safety' error checks.

frplacd[x;y]

Has the same definition as rplacd but compiles open as one instruction. Note that no checks are made on x, so that a compiled frplacd can clobber NIL, producing strange and wondrous effects.

frplaca[x;y]	Similar to *frplacd*.
quote[x]	This is a function that prevents its argument from being evaluated. Its value is x itself.
kwote[x]	(LIST (QUOTE QUOTE) X), if x=A, y=B, (KWOTE (CONS X Y)) = (QUOTE (A . B)).
cond[c_1;c_2;...;c_k]	The conditional function of LISP, *cond*, takes an indefinite number of arguments $c_1, c_2, \dots c_k$, called clauses. Each clause c_i is a list $(e_{1i} \dots e_{ni})$ of $n \geq 1$ items. The clauses are considered in sequence as follows: the first expression e_{1i} of the clause c_i is evaluated and its value is classified as *false* (equal to NIL) or *true* (not equal to NIL). If the value of e_{1i} is *true*, the expressions $e_{2i} \dots e_{ni}$ that follow in clause c_i are evaluated in sequence, and the value of the conditional is the value of e_{ni}, the last expression in the clause. In particular, if n=1, i.e., if there is only one expression in the clause c_i, the value of the conditional is the value of e_{1i}. (which is evaluated only once). If e_{1i} is false, then the remainder of clause c_i is ignored, and the next clause c_{i+1} is considered. If no e_{1i} is true for *any* clause, the value of the conditional expression is NIL. See p. 4.3 for an example.

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

This very useful function is used to select a sequence of instructions based on the value of its first argument $\underline{x}$. Each of the $\underline{y}_i$ is a list of the form ($\underline{s}_i$ $\underline{e}_{1i}$ $\underline{e}_{2i}$...$\underline{e}_{ki}$) where $\underline{s}_i$ is the selection key.

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

If $\underline{s}_i$ is a list, and if any element (not evaluated) 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))
         (COND
           ((NULL X)
             NIL)
           (T (QUOTE STOP]
```

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

selectq compiles open, and is therefore very fast; however, it will not work if the value of *x* is a list, a large integer, or floating point number, since it uses *eq*.

$\mathrm{prog1}[x_1;x_2;\ldots;x_n]$

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

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

progn evaluates each of its arguments in sequence, and returns the value of its last argument as its value. *progn* is used to specify more than one computation where the syntax allows only one, e.g.
(SELECTQ .... (PROGN ...))
allows evaluation of several expressions as the default condition for a *selectq*.

$\mathrm{prog}[args;e_1;e_2;\ldots;e_n]$

This feature allows the user to write an ALGOL-like program containing LISP statements to be executed. The first 'argument' is a list of program variables. (Must be NIL if no variables are used). Each atom in this list is bound to NIL. Each list must be of the form (atom form). _atom_ is bound to the value of _form_, the evaluation taking place before any bindings, e.g.,

```
(PROG ((X Y) (Y X)) ...)
```

will bind _x_ to the value of _y_ and _y_ to the (original) value of _x_.

The rest of the _prog_ is a sequence of (non-atomic) statements (forms) and atomic symbols used as labels for _go_. The forms are evaluated sequentially, with labels being skipped. The two special functions _go_ and _return_ alter this flow of control as described below. The value of the _prog_ is usually specified by the function _return_. If no _return_ is executed, i.e., if the prog "falls off the end," the value of the _prog_ is undefined, i.e. garbage.

go[x]

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

return[x]

A return is the normal exit for a prog. Its argument is evaluated and is the value of the prog in which it appears.

If a go or return is executed in an interpreted function which is not a prog, the go or return will be executed in the last interpreted *prog entered if any, otherwise cause an error.*

go or return inside of a compiled function that is not a prog is not allowed, and will cause an error at compile time.

As a corollary, go or return in a functional argument, e.g. to mapc, will not work compiled. Also, since nlsetq's and ersetq's compile as *separate* functions, a go or return *cannot* be used inside of a compiled nlsetq or ersetq if the corresponding prog is outside, i.e. above, the nlsetq or ersetq.

set[x;y]

This function sets x to y. Its value is y. If x is not a literal atom, or x is NIL, causes an error. Note that set is a normal lambda-spread function, i.e., its arguments are evaluated before it is called. Thus, if the value of x is c, and the value of y is b, then set[x;y] would result in c having value b, and b being returned as the value of set.

setq[x;y]	An nlambda version of set: the first argument is not evaluated. Thus if the value of x is c and the value of y is b, setq[x;y] would result in x (not c) being set to b, and b being returned. If x is not a literal atom, or x is NIL, an error is generated.
setqq[x;y]	Identical to setq except that neither argument is evaluated. Thus setqq[x;y] sets X to Y.

Predicates and Logical Connectives

atom[x]	is T if x is an atom; NIL otherwise.
litatom[x]	is T is x is a literal atom, i.e., not a number, NIL otherwise.
numberp[x]	is x if x is a number, NIL otherwise.

Convention: Functions that end in p are frequently predicates, i.e. they test for some condition.

stringp[x]	is x if x is a string, NIL otherwise.*
arrayp[x]	is x if x is an array, NIL otherwise.

*For other string functions, see Section 10.

listp[x]	is x if x is a nonatomic list-structure, i.e., one created by one or more conses; NIL otherwise. Note that arrays and strings are not atoms, but are not lists.
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 36 bit compare of pointers. Its value is not guaranteed T for equal numbers which are not small integers. See eqp.
neq[x;y]	The value of neq is T if x is not eq to y, and NIL otherwise.
null[x]	eq[x;NIL]
not[x]	same as null, that is eq[x;NIL].
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.*
equal[x;y]	The value of this function is T if x and y print identically; the value of equal is NIL otherwise. Note that x and y do *not* have to be eq.

*For other number functions, see Section 13.

and[x_1;x_2;...;x_n]	Takes an indefinite number of arguments (including ∅). If all of its arguments have non-null value, its value is the value of its last argument, otherwise NIL. E.g. and[x;member[x;y]] will have as its value either NIL or a tail of y. and[]=T. Evaluation stops at the first argument whose value is NIL.
or[x_1;x_2;...;x_n]	Takes an indefinite number of arguments (including ∅). Its value is that of the first argument whose value is not NIL, otherwise NIL if all arguments have value NIL. e.g. or[x;numberp[y]] has its value x, y, or NIL. or[]=NIL. Evaluation stops at the first argument whose value is not NIL.
every[everyx;everyf]	Is T if the result of applying everyf to each element in everyx is true, otherwise NIL. E.g., every[(X Y Z); ATOM]=T.
some[somex;somef]	is the tail of somex beginning with the first element that satisfies somef, i.e., for which somef applied to that element is true. Value is NIL is no such element exists. E.g., some[x;(LAMBDA (Z) (EQUAL Z Y))] is equivalent to member[y;x].
notany[somex;somef]	not[some[somex;somef]]
notevery[everyx;everyf]	not[every[everyx;everyf]]

memb[x;y]	Determines if x is a member of list y, i.e., if there is an element of y eq to x. If so, its value is the tail of the list y starting with that element. If not, its value is NIL.
fmemb[x;y]	Fast version of memb that compiles open as a five instruction loop, terminating on a NULL check.
member[x;y]	Identical to memb except that it uses equal instead of eq to check membership of x in y.

COMMENT: EQ VS EQUAL: The reason for the existence of both memb and member is that eq compiles as one instruction but equal requires a function call, and is therefore considerably more expensive. Wherever possible, the user should write (and use) functions that use eq instead of equal.

tailp[x;y]	Is x, if x is a list and a *tail* of y, i.e., x is eq to some number of cdrs ≥Ø[*] of y, NIL otherwise.
assoc[x;y]	y is a list of lists (usually dotted pairs). The value of assoc is the first sublist of y whose car is eq to x. If such a list is not found, the value is NIL. Example: assoc[B;((A . 1)(B .2)(C . 3))]=(B . 2).
fassoc[x;y]	Fast version of assoc that compiles open as a 6 instruction loop, terminating on a NULL check.
sassoc[x;y]	Same as assoc but uses equal instead of eq.

[*]If x is eq to some number of cdrs ≥1 of y, we say x is a proper tail (of y).

SECTION VI

LIST MANIPULATION AND CONCATENATION

Contents

list[x_1;x_2;...;x_n]

lambda-nospread function. Its value is a list of the values of its arguments.

append[x_1;x_2;...;x_n]

Copies the top level of the list $\underline{x}_1$ and appends this to a copy of top level list $\underline{x}_2$ appended to ... appended to $\underline{x}_n$, e.g.

```
append[(A B) (C D E) (F G)] =
                  (A B C D E F G)
```

Note that only the first n-1 lists are copies. However n=1 is treated specially; i.e. append[x] can be used to copy the top level of a single list.*

The following examples illustrate the treatment of non-lists.

```
append[(A B C);D] = (A B C . D)
append[A;(B C D)] = (B C D)
append[(A B C . D);(E F G)] =
                  (A B C E F G)
append[(A B C . D)] = (A B C . D)
```

*To copy a list to all levels, use copy.

nconc[x_1;x_2;...;x_n]	Returns same value as <u>append</u> but actually modifies the list structure of x_1 ... x_{n-1}.
nconc1[lst;x]	Performs nconc[lst;list[x]]. The <u>cons</u> will be on the same page as <u>lst</u>.
tconc[ptr;x]	<u>tconc</u> is useful for building a list by adding elements one at a time at the end. i.e. its role is similar to that of <u>nconc1</u> However, unlike <u>nconc1</u>, <u>tconc</u> does not have to search to the end of the list each time it is called. It does this by keeping a pointer to the end of the list being assembled, and updating this pointer after each call. The savings can be considerable for long lists. The cost is the extra word required for storing both the list being assembed, and the end of the list. <u>ptr</u> is that word: car[ptr] is the list being assembled, cdr[ptr] is last[car[ptr]]. The value of <u>tconc</u> is <u>ptr</u>, with the appropriate modifications to <u>car</u> and <u>cdr</u>. Example:

```
(RPTQ 5 (SETQ FOO (TCONC FOO RPTN)))
((5 4 3 2 1) 1)
```

<u>tconc</u> can be initialized in two ways. If <u>ptr</u> is NIL, <u>tconc</u> will make up a ptr. In this case, the program must set some variable to the value of the first call to <u>tconc</u>. After that, it

is unnecessary to reset since tconc physically changes ptr. Thus

```
(SETQ FOO (TCONC NIL 1))
((1) 1)
(RPTQ  4 (TCONC FOO RPTN))
((1 4 3 2 1) 1)
```

If ptr is initially (NIL), the value of tconc is the same as for ptr=NIL, but tconc changes ptr, e.g.

```
(SETQ FOO (CONS))
(NIL)
(RPTQ 5 (TCONC FOO RPTN))
((5 4 3 2 1) 1)
```

The latter method allows the program to initialize, and then call tconc without having to perform setq on its value.

lconc[ptr;x]

Where tconc is used to add *elements* at the end of a list, lconc is used for building a list by adding *lists* at the end, i.e. it is similar to nconc instead of nconc1, e.g.

```
(SETQ FOO (CONS))
(NIL)
(LCONC FOO (LIST 1 2))
((1 2)2)
(LCONC FOO (LIST 3 4 5))
((1 2 3 4 5) 5)
(LCONC FOO NIL)
((1 2 3 4 5) 5)
```

Note that

```
(TCONC FOO NIL))
((1 2 3 4 5 NIL) NIL)
(TCONC FOO (LIST 3 4 5))
((1 2 3 4 5 NIL (3 4 5)) (3 4 5))
```

lconc uses the same pointer conventions as tconc for eliminating searching to the end of the list, so that the same pointer can be given to tconc and lconc interchangeably.

attach[x;y]

Value is equal to cons[x;y], but attaches x to the front of y by doing an rplaca and rplacd, i.e. the value of attach is eq to y, which it physically changes. y must be a list or an error is generated.

remove[x;l]

Removes all occurrences of x from list l, giving a copy of l with all elements equal to x removed.

CONVENTION: naming a function by prefixing an existing function with d frequently indicates the new function is a destructive version of the old one, i.e. it does not make any new structure but cannibalizes its argument(s).

dremove[x;l]

Similar to remove, but uses eq instead of equal, and actually modifies the list l when removing x, and thus does not use any additional storage. More efficient than remove.

copy[x]

Makes a copy of the list x. The value of copy is the copied list. All levels of x are copied, down to non-lists, i.e. if x contains arrays and strings the copy of x will contain the identical arrays and strings. Copy is recursive in the car direction only, so that very long lists can be copied. Note: to copy just the *top* level of x, do append[x].

reverse[l]

Reverses (and copies) the top level of a list, e.g.
reverse[(A B (C D))] = ((C D) B A)
If x is not a list, value is x.

dreverse[l]	Value is same as that of *reverse*, but *dreverse* destroys the original list *l* and thus does not use any additional storage. More efficient than *reverse*.
subst[x;y;z]	Value is 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 subexpression of *z* or when *y* is both atomic and *eq* to *cdr* of some subexpression of *z*. For example: subst[A;B;(C B (X . B))] = (C A (X . A)) subst[A;(B C);((B C) D B C)] = (A D B C), not (A D . A) The value of *subst* is a copy of *z* with the appropriate changes. Furthermore, if *x* is a list, it is copied at each substitution.
dsubst[x;y;z]	Similar to *subst*, but uses *eq* and does not copy *z*, but changes the list structure *z* itself. Like *subst*, *dsubst* substitutes with a copy of *x*. More efficient than *subst*.
lsubst[x;y;z]	Like *subst* except *x* is substituted as a segment, e.g. lsubst[(A B); Y; (X Y Z)] is (X A B Z). Note that if *x* is NIL, produces a copy of *z* with all *y*'s deleted.

esubst[x;y;z;flg]

Similar to dsubst, but first checks to see if y actually appears in z. If not, calls error! where flg=T means print a message of the form x ?. This function is actually an implementation of the editor's R command (see Section 9), so that y can use &, --, or alt-modes a la the R command.

sublis[alst;expr;flg]

alst is a list of pairs: $((u_1 . v_1) (u_2 . v_2) ... (u_n . v_n))$ with each u_i atomic.

The value of sublis[alst;expr;flg] is the result of substituting each v for the corresponding u in expr.*
Example:

```
  sublis[((A . X)(C . Y));(A B C D)]=
                    (X B Y D)
```

New structure is created only if needed or if flg=T, e.g. if flg=NIL and there are no substitutions, value is eq to expr.

subpair[old;new;expr;flg]

Similar to sublis, except that elements of new are substituted for corresponding atoms of old in expr. Example:

```
  subpair[(A C);(X Y);(A B C D)]=
               (X B Y D)
```

As with sublis, new structure is created only if needed, or if flg=T, e.g. if flg=NIL and there are no substitutions, the value is eq to expr.

*To remember the order on alst think of it as old to new, i.e. $u_i \rightarrow v_i$.

Note that subst, dsubst, lsubst, and esubst all substitute copies of the appropriate expression, whereas subpair and sublis substitute the identical structure (unless flg=T).

last[x]	Value is a pointer to the last cell in the list x, e.g. if x=(A B C) then last[x]=(C). If x=(A B . C) last[x] = (B . C). Value is NIL if x is not a list.
flast[x]	Fast version of last that compiles open as a 5 instruction loop, terminating on a NULL check.
nleft[l;n]	Value is last n elements of l. Value is NIL if l is not a list of length ≥ n.
lastn[l;n]	Value is cons[x;y] where y is the last n elements of l, and x is the initial segment. e.g. lastn[(A B C D E);2]=((A B C) D E) lastn[(A B);2]=(NIL A B) Value is NIL if l is not a list containing at least n elements.
nth[x;n]	Value is the tail of x beginning with the nth element, e.g. if n=2, value is cdr[x], if n=3, cddr[x], etc. If n=1, value is x, if n=0, for consistency, value is cons[NIL;x]
fnth[x;n]	Fast version of nth that compiles open as a 3 instruction loop, terminating on a NULL check.

length[x]	Value is the length of the list x where length is defined as the number of cdrs required to reach a nonlist, e.g. length[(A B C)] = 3 length[(A B C . D)] = 3 length[A] = 0
flength[x]	Fast version of length that compiles open as a 4 instruction loop, terminating on a NULL check.
count[x]	Value is the number of list words in the structure x. Thus, count is like a length that goes to all levels. Count of a non-list is 0.
ldiff[x;y;z]	y must be a tail of x, i.e. eq to 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;member[FOO;x]] gives all elements in x up to the first FOO. Note that the value of ldiff is always new list structure unless y=NIL, in which case ldiff[x;NIL] is x itself. 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. If y is not a tail of x, generates an error. ldiff terminates on a null check.

intersection[x;y]	Value is a list whose elements are members of both lists x and y. Note that intersection[x;x] gives a list of all members of x without any duplications.
union[x;y]	Value is a (new) list consisting of all elements included on either of the two original lists. It is more efficient to make x be the shorter list.*
sort[data;comparefn]	data is a list of items to be sorted using comparefn, a predicate function of two arguments which can compare any two items on data and return T if the first one belongs before the second. If comparefn is NIL, alphorder is used; thus sort[data] will alphabetize a list. If comparefn is T, car's of items are given to alphorder; thus sort[a-list;T] will alphabetize by the car of each item. sort[x;ILESSP] will sort a list of integers. The value of sort is the sorted list. The sort is destructive and uses no extra LISP data space. The value returned is eq to data but elements have been switched around. Interrupting with control D, E, or B

*The value of union is y with all elements of x not in y consed on the front of it. Therefore, if an element appears twice in y, it will appear twice in union[x;y]. Also, since

```
        union[(A) ; (A A)] = (A A)
but     union[(A A) ; (A)] = (A)
```

union is non-commutative.

may cause loss of data, but control H may be used at any time, and sort will break at a clean state from which ↑ or control characters are safe. The algorithm has been optimized with respect to the number of compares.

Note that if comparefn[a;b] = comparefn[b;a] then the ordering of a and b may or may not be preserved. For example, if (FOO . FIE) appears before (FOO . FUM) in x, sort[x;T] may or may not reverse the order of these two elements. Of course, the user can always specify a more precise comparefn, e.g.

```
        [LAMBDA (X Y)
                (COND ((EQ (CAR X)(CAR Y)) (ALPHORDER (CDR X) (CDR Y)))
                      (T (ALPHORDER (CAR X) (CAR Y]
```

merge[a;b;comparefn]

a and b are lists which have previously been sorted using sort and comparefn. Value is a destructive merging of the two lists. It does not matter which list is longer. After reversing both a and b are eq to the merged list. merge may be aborted after control H.

alphorder[a;b]

A predicate function of two arguments, for alphabetizing. Returns T if its first argument belongs before its second. Numbers come before literal atoms, and are ordered by magnitude (using greaterp). Literal atoms and strings are ordered by comparing the (ASCII) character codes in their pnames. Thus alphorder[23;123] is T, whereas alphorder[A23;A123] is NIL, because the character code for the digit 2 is greater than the code for 1.

Atoms and strings are ordered before all other data types. If neither _a_ nor _b_ are atoms or strings, the value of _alphorder_ is T, i.e. in order. Note: **_alphorder_** does no **_unpacks_**, **_chcons_**, **_conses_**, or **_nthchars_**. It is several times faster for alphabetizing than anything that can be written using these other functions.

SECTION VII

PROPERTY LISTS AND HASH LINKS

Contents

Property Lists

Property lists are entities associated with literal atoms, which are stored on cdr of the atom. Property lists are **conventionally** *lists of the form (property value property value ... property value) although the user can store anything he wishes in cdr of a literal atom. However, the functions which manipulate property lists observe this convention by cycling down the property list two cdrs at a time. Similarly, most of these functions generate an error if given an argument which is not a literal atom, i.e., they cannot be used directly on lists.*

The term 'property name' or 'property' is used for the property indicators appearing in the odd positions, and the term 'property value' or 'value of a property' or simply 'value' for the values appearing in the even positions. Sometimes the phrase 'to store on the property --' is used, meaning to place the indicated information on the property list under the property name --.

Properties are usually atoms, although no checks are made to eliminate use of non-atoms in an odd position. However, the property list searching functions all use eq.

Property List Functions

put[atm;prop;val]

This function puts on the property list of atm, the property prop with value val. val replaces any previous value for the property prop on this property list. Generates an error if atm is not a literal atom. Value is val.

addprop[atm;prop;new;flg]	This function *adds* the value *new* to the list which is the value of property *prop* on property list of *atm*. If *flg* is T, *new* is *consed* onto the front of value of *prop*, otherwise it is *nconced* on end (*nconcl*). If *atm* does not have a *prop*, the effect is the same as put[atm;prop;list[new]]. e.g. if addprop[FOO;PROP;FIE] is followed by addprop[FOO;PROP;FUM], getp[FOO;PROP] will be (FIE FUM). The value of *addprop* is the (new) property value. If *atm* is not a literal atom, an error occurs.
remprop[atm;prop]	This function removes all occurrences of the property *prop* (and its value) from the property list of *atm*. Value is *prop* if any were found, otherwise NIL. If *atm* is not a literal atom, an error occurs.
changeprop[x;propl;prop2]	Changes *name* of property *propl* to *prop2* on property list of *x*, (but does not affect the value of the property). Value is *x*, unless *propl* is not found, in which case, the value is NIL. If *x* is not a literal atom, an error occurs.
get[x;y]	Gets the item after the atom *y* on list *x*. If *y* is not on the list *x*, value is NIL. For example, get[(A B C D);B]=C. Note that since *get* terminates on a non-list, get[atom,anything] is NIL. Therefore, to search a property list, *getp* should be used, or *get* applied to cdr[atom].

getp[atm;prop]

This function gets the property value for prop from the property list of atm. The value of getp is NIL if atm is not a literal atom, or prop is not found. Note that the value may also be NIL if the property value is NIL

Note: Since getp searches a list two items at a time, the same object can be both a property and a value. e.g., if the property list of atm is (PROP1 A PROP2 B A C)

```
      getp[atm;A] = C.
```

Note however that

```
get[cdr[atm];A] = PROP2
```

getlis[atm;props]

props is a list of properties. getlis searches the property list of atm two cdrs at a time, and returns the property list as of the first property on props that it finds E.g., if the property list of atm is (PROP1 A PROP3 B A C)

```
getlis[atm;(PROP2 PROP3)]=(PROP3 B A C).
```

Value is NIL is atm not a literal atom or no properties found.

deflist[l;prop]

This function is used to put values under the same property name on the property lists of several atoms. l is a list of two-element lists. The first element of each is a literal atom, and the second element is the property value for the property prop. The value of deflist is NIL.

Note: Many atoms in the system already have property lists, ususally for use by the compiler. Be careful not to clobber their property lists by using rplacd. The value of sysprops is a list of the property names used by the system.

Hash Links

The description of the hash link facility in BBN-LISP is included in the chapter on property lists because of the similarities in the ways the two features are used. A property list provides a way of associating information with a particular atom. A hash link is an association between any LISP pointer (atoms, numbers, arrays, strings, lists, et al) called the hash-item, and any other LISP pointer called the hash-value. Property lists are stored in cdr of the atom. Hash links are implemented by computing an address, called the hash-address, in a specified array, called the hash-array, and storing the hash-value and the hash-item into the cell with that address. The contents of that cell, i.e. the hash-value and hash-item, is then called the hash-link.*

Since the hash-array is obviously much smaller than the total number of possible hash-items,** the hash-address computed from item may already contain a hash-link. If this link is from item,*** the new hash-value simply replaces the old hash-value. Otherwise, another hash-address (in the same hash-array) must be computed, etc, until an empty cell is found,**** or a cell containing a hash-link from item.

*The term hash link (unhypehnated) refers to the process of associating information this way, or the 'association' as an abstract concept.

**which is the total number of LISP pointers, i.e., 256K.

***eq is used for comparing item with the hash-item in the cell

****After a certain number of iterations (the exact algorithm is complicated), the hash-array is considered to be full, and the array is either enlarged, or an error is generated, as described below in the discussion of overflow.

When a hash link for item is being retrieved, the hash-address is computed using the same algorithm as that employed for making the hash link. If the corresponding cell is empty, there is no hash link for item. If it contains a hash-link from item, the hash-value is returned. Otherwise, another hash-address must be computed, and so forth.*

Note that more than one hash link can be attached to a given hash-item by using more than one hash-array.

Hash Link Functions

In the description of the functions below, the argument array has one of three forms: (1) NIL, in which case the hash-array provided by the system, syshasharray, is used;** (2) a hash-array created by the function harray, or created from an ordinary array using clrhash as described below; or (3) a list car of which is a hash-array. The latter form is used for specifying what is to be done on overflow, as described below.

harray[n]	creates a hash-array of size n, equivalent to clrhash[array[n]].
clrhash[array]	sets all elements of array to Ø and sets left half of first word of header to -1.
puthash[item;val;array]	puts into array a hash-link from item to val. Replaces previous link from same item, if any. If val=NIL any old link is removed, (hence a hash-value of NIL is not allowed).

*For reasonable operation, the hash array should be ten to twenty percent larger than the maximum number of hash links to be made to it.

** syshasharray is not used by the system, it is provided solely for the user's benefit. It is initially 512 words large, and is automatically enlarged by 50% whenever it is 'full'. See p. 7.7.

gethash[item;array]	finds hash-link from *item* in *array* and returns the hash-value. Value is NIL if no link exists.
rehash[oldar;newar]	hashes all items and values in *oldar* into *newar*. The two arrays do not have to be (and usually aren't) the same size. Value is *newar*.
maphash[array;maphfn]	*maphfn* is a function of two arguments. For each hash-link in array, *maphfn* will be applied to the hash-value and hash-item, e.g. maphash[array;(LAMBDA (X Y) (AND (LISTP Y) (PRINT X)))] will print the hash-value for all hash-links from lists. The value of *maphash* is *array*.
dmphash[arrayname]	Nlambda nospread that prints on the primary output file a *load*able form which will restore what is in the array specified by *arrayname*, e.g. (E (DMPHASH SYSHASHARRAY)) as a *prettydef* command will dump the system hash-array.

Note that all *eq* identities except atoms and small integers are lost by dumping and loading because new conses are done for each item. Thus if two lists contain an *eq* substructure, when they are dumped and loaded back in, the corresponding substructures, while *equal* are no longer *eq*.

Hash Overflow

The user can provide for automatic enlargement of a hash-array when it overflows, i.e., is full and an attempt is made to store a hash link into it, by using an array argument of the form (hash-array . n), _n_ a positive integer; (hash-array . f), _f_ a floating point number; or (hash-array). In the first case, a new hash-array is created with _n_ more cells than the current hash-array. The old array is then rehashed into the new hash-array, the new hash-array is _rplaca_ed into the dotted pair, and the computation continues. In the second case, the new hash array will be _f_ times the size of the current hash-array. The third case, (hash-array), is equivalent to (hash-array . 1.5).

If a hash array overflows, and the array argument used was not one of these three forms, an error is generated, HASH TABLE FULL, which will either cause a break or unwind to the last _errorset_ as per treatment of errors described in Section 16.

The system hash array, _syshasharray_, is automatically enlarged by 1.5 when it is full.

SECTION VIII

FUNCTION DEFINITION AND EVALUATION

Contents

General Comments

A function definition in LISP is stored in a special cell associated with each literal atom called the function definition cell. This cell is directly accessible via the two functions putd, which puts a definition in the cell, and getd which gets the definition from the cell. In addition, the function fntyp returns the function type, i.e., EXPR, EXPR* ... FSUBR* as described in chapter 4. exprp, ccodep, and subrp are true if the function is an expr, compiled function, or subr respectively; argtype returns Ø, 1, 2, or 3 depending on whether the function is a spread or nospread (i.e., its fntyp ends in *), or evaluate or no-evaluate (i.e., its fntyp begins with F or CF); arglist returns the list of arguments; and nargs returns the number of arguments. fntyp, exprp, ccodep, subrp, argtype, arglist, and nargs can be given either a literal atom, in which case they obtain the function definition from the atom's definition cell, or a function definition itself.

Subrs

Because subrs,* are called in a special way, their definitions are stored differently than those of compiled or interpreted functions. In the right half of the definition cell is the address of the first instruction of the subr, and in the left half its argtype: 0, 1, 2, 3. getd of a subr returns a dotted pair or argtype and address. This is not the same word as appears in the definition cell, but a new cons; i.e., each getd of a subr performs a cons. Similarly, putd of a definition of the form (number . address), where number = 0, 1, 2, or 3, and address is in the appropriate range, stores the definition as a subr, i.e., takes the cons apart and stores car in the left half of the definition cell and cdr in the right half.

Validity of Definitions

Although the function definition cell is intended for function definitions, putd and getd do not make thorough checks on the validity of definitions that "look like" exprs, compiled code, or subrs. Thus if putd is given an array pointer, it treats it as compiled code, and simply stores the array pointer in the definition cell. getd will then return the array pointer. Similarly, a call to that function will simply transfer to what would normally be the entry point for the function, and produce random results if the array were not a compiled function.

Similarly, if putd is given a dotted pair of the form (number . address) where number is 0, 1, 2, or 3 and address falls in the

Basic functions, handcoded in machine language. e.g. cons, car, cond. The term subrs includes spread/nospread, eval/noeval functions, i.e. the four fntyp's subr, fsubr, subr, and fsubr*.

subr range, putd assumes it is a subr and stores it away as described earlier. getd would then return cons of the left and right half, i.e., a dotted pair equal (but not eq) to the expression originally given putd. Similarly, a call to this function would transfer to the corresponding address.

Finally, if putd is given any other list, it simply stores it away. A call to this function would then go through the interpreter as described in the appendix.

Note that putd does not actually check to see if the s-expression is a valid definition, i.e., begins with LAMBDA or NLAMBDA. Similarly, exprp is true if a definition is a list and not of the form (number . address), number = 0, 1, 2, or 3 and address a subr address; subrp is true if it is of this form. arglist and nargs work correspondingly.

Only fntyp and argtype check function definitions further than that described above: both argtype and fntyp return NIL when exprp is true but car of the definition is not LAMBDA or NLAMBDA.* In other words, if the user uses putd to put (A B C) in a function definition cell, getd will return this value, the editor and prettyprint will both treat it as a definition, exprp will return T, ccodep and subrp NIL, arglist B, and nargs 1.

* These functions have different value on LAMBDAs and NLAMBDAs and hence must check. The compiler and interpreter also take different actions for LAMBDAs and NLAMBDAs, and therefore generate errors if the definition is neither.

getd[x]	gets the function definition of x. Value is the definition. Value is NIL if x is not a literal atom, or has no definition.
putd[x;y]	puts the definition y into x's function cell. Value is y. Gives an error if x is not a literal atom, or y is a string, number, or literal atom other than NIL.
putdq[x;y]	nlambda version of putd; both arguments are considered quoted. 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. copyflg=T is only meaningful for exprs, although movd works for compiled code and subrs. The value of movd is to.

NOTE: fntyp, subrp, ccodep, exprp, argtype, nargs, and arglist all can be given either the name of a function, or a definition.

fntyp[fn]	Value is NIL if fn is not a function definition or the name of a defined function. Otherwise fntyp returns one of the following as defined in the section on function types:

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

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

subrp[fn]	is true if and only if fntyp[fn] is either SUBR, FSUBR, SUBR*, or FSUBR*, i.e., the third column of fntyp's
ccodep[fn]	is true if and only if fntyp[fn] is either CEXPR, CFEXPR, CEXPR*, or CFEXPR*, i.e., second column of fntyp's
exprp[fn]	is true if fntyp[fn] is either EXPR, FEXPR, EXPR*, or FEXPR*, i.e., first column of fntyp's. However, exprp[fn] is also true if fn is (has) a list definition that is not a SUBR, but does not begin with either LAMBDA or NLAMBDA. In other words, exprp is not quite as selective as fntyp.

argtype[fn]

fn is the name of a function or its definition. The value of argtype is the argtype of fn, i.e., Ø, 1, 2, or 3, or NIL if fn is not a function. The interpretation of the argtype is:

0 eval/spread function
(EXPR, CEXPR, SUBR)

1 no-eval/spread functions
(FEXPR, CFEXPR, FSUBR)

2 eval/nospread functions
(EXPR*, CEXPR*, SUBR*)

3 no-eval/nospread functions
(FEXPR*, CFEXPR*, FSUBR*)

i.e., argtype corresponds to the *rows* of fntyps.

nargs[fn]

value is the number of arguments of fn, or NIL if fn is not a function.* nargs uses exprp, not fntyp, so that nargs[(A (B C) D)]=2. Note that if fn is a SUBR or FSUBR, nargs = 3, regardless of the number of arguments logically needed/used by the routine. If fn is a nospread function, nargs=1.

arglist[fn]

value is the 'argument list' for fn. Note that the 'argument list' is an atom for nospread functions. Since NIL is a possible value for arglist an an error is generated if fn is not a function.*

If fn is a SUBR or FSUBR, the value of arglist is (U V W), if a SUBR* or FSUBR*, the value is U. This is merely a 'feature' of arglist, subrs do not actually store the names u, v, or w on the stack. However, if the user breaks or traces a subr (Section 15), these will be the argument names used when an equivalent expr definition is constructed.

define[x]

The argument of define is a list. Each element of the list is itself a list either of the form (*name definition*) or (*name arguments* ...). In the second case, following *arguments* is the body of the definition. 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.

*i.e., if exprp, ccodep, and subrp are all NIL.

defineq[x_1;x_i;...;x_n]	*nlambda* nospread version of *define*, i.e., takes an indefinite number of arguments which are not evaluated. Each x_i must be a list, of the form described in *define*. *defineq* calls *define*, so *dfnflg* affects its operation the same as *define*.
savedef[fn]	Saves the definition of *fn* on its property list under property EXPR, CODE, or SUBR depending on its *fntyp*. Value is the property name used. If getd[fn] is non-NIL, but fntyp[fn] is NIL, saves on property name LIST. This situation can arise when a function is redefined which was originally defined with LAMBDA misspelled or omitted. If *fn* is a list, *savedef* operates on each function in the list, and its value is a list of the individual values.

unsavedef[fn;prop]	Restores the definition of _fn_ from its property list under property _prop_ (see _savedef_ above). Value is _prop_. If nothing saved under _prop_, and _fn_ is defined, returns (prop NOT FOUND), otherwise generates an error.
	If _prop_ is *not* given, _unsavedef_ looks under EXPR, CODE, and SUBR, in that order. The value of _unsavedef_ is the property name, or if nothing is found and _fn_ is a function, the value is (NOTHING FOUND); otherwise an error occurs. If _dfnflg_=T, the current definition of _fn_, if any, 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 definition cell.
	If _fn_ is a list, _unsavedef_ operates on each function of the list, and its value is a list of the individual values.

eval[x]*

eval evaluates the expression x and returns this value. Note that eval is itself a lambda type function, so *its* argument is first evaluated, e.g.,

```
←SET(FOO (ADD1 3))
 (ADD1 3)
←(EVAL FOO)
 4
←EVAL(FOO)
 (ADD1 3)
```

e[x]

nlambda nospread version of eval. Thus it eliminates the extra pair of parentheses for the list of arguments for eval. i.e., e x is equivalent to eval[x]. Note however that in BBN-LISP, the user can type just x to get x evaluated. See page 2.4.

*eval is a subr so that the 'name' x does not actually appear on the stack.

apply[fn;args]

<u>apply</u> applies the function <u>fn</u> to the arguments <u>args</u>, i.e. the individual elements of <u>args</u> are not evaluated by <u>apply</u>. However like <u>eval</u>, <u>apply</u> is a <u>lambda</u> function so *its* arguments are evaluated before it is called e.g.,

```
←SET(FOO1 3)
 3
←SET(FOO2 4)
 4
←(APPLY (QUOTE IPLUS) (LIST FOO1 FOO2))
 7
```

Here, <u>foo1</u> and <u>foo2</u> were evaluated when the second argument to <u>apply</u> was evaluated. Compare with

```
←SET(FOO1 (ADD1 2))
 (ADD1 2)
←SET(FOO2 (SUB1 5))
 (SUB1 5)
←(APPLY (QUOTE IPLUS) (LIST FOO1 FOO2))
 NON-NUMERIC ARG
 (ADD1 2)
```

Note that:

```
←(EVAL (LIST (QUOTE IPLUS) FOO1 FOO2))
 7
```

because

```
←EVAL((LIST (QUOTE IPLUS) FOO1 FOO2))
 (IPLUS (ADD1 2) (SUB1 5))
```

evala[x;a]

Simulates a-list evaluation as in LISP 1.5. <u>x</u> is a form, <u>a</u> is a list of dotted pairs of variable name and value. <u>a</u> is 'spread' on the stack, and then <u>x</u> is evaluated, i.e., any variables appearing free in <u>x</u>, that also appears as <u>car</u> of an element of <u>a</u> will be given the value in the <u>cdr</u> of that element.

rpt[rptn;rptf]

Evaluates the expression rptf rptn times. At any point, rptn is the number of evaluations yet to take place. Returns the value of the last evaluation. If rptn ≤ Ø, rptf is not evaluated, and the value of rpt is NIL.

NOTE: rpt is a lambda function, so both its arguments are evaluated before rpt is called. For most applications, the user will probably want to use rptq.

rptq[rptn;rptf]

nlambda version of rpt: rptn is evaluated, rptf quoted.

arg[var;m]

Used to access the individual arguments of a lambda nospread function. arg is an nlambda function used like setq: var is the *name* of the atomic argument list, and is considered to be quoted, m is the number of the desired argument, and is evaluated. For example, consider the following definition of iplus in terms of plus.

```
[LAMBDA X
  (PROG ((M 0)
         (N 0))
    LP  (COND
          ((EQ N X)
            (RETURN M)))
        [SETQ M (PLUS M (ARG X (SETQ N (ADD1 N]
        (GO LP]
```

The value of arg is undefined for m less than or equal to 0 or greater than the *value* of var.* Lower numbered arguments appear earlier in the form, e.g. for (IPLUS A B C), arg[X;1]=the value of A, arg[X;2]=the value of B, and arg[X;3]=the value of C. Note that the lambda variable should *never* be reset. However, individual arguments can be reset using setarg described below.

* For lambda nospread functions, the lambda variable is bound to the number of arguments actually given to the function. See Section 4.

setarg[var;m;x]

sets to x the mth argument for the lambda nospread function whose argument list is var. var is considered quoted, m and x are evaluated; e.g. in the previous example, (SETARG X (ADD1 N) (MINUS M)) would be an example of the correct form for setarg.

SECTION IX

THE LISP EDITOR

Contents

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(FOO). However, the editor can also be used to edit the value of a variable, via editv, to edit a property list, via editp, or to edit an arbitrary expression, via edite. It is an important feature which allows good on-line interaction in the BBN LISP system.

This chapter begins with a lengthy introduction intended for the new user. The reference portion begins on page 9.17.

Introduction

Let us introduce some of the basic editor commands, and give a flavor for the editor's language structure by guiding the reader through a hypothetical editing session. Suppose we are editing the following incorrect definition of append

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

We call the editor via the function editf:

```
←EDITF(APPEND)
EDIT
*
```

The editor responds by typing EDIT followed by *, which is the editor's ready character, i.e., it signifies that the editor is ready to accept commands.[†]

At any given moment, the editor's attention is centered on some substructure of the expression being edited. This substructure is called the *current expression*, and it is what the user sees when he gives the editor the command P, for print. Initially, the current expression is the top level one, i.e., the entire expression being edited. Thus:

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

[†] In other words, all lines beginning with * were typed by the user, the rest by the editor.

Note that the editor prints the current expression as though printlevel were set to 2, i.e., sublists of sublists are printed as &. The command ? will print the current expression as though printlevel were 1000

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

and the command PP will prettyprint the current expression.

A positive integer is interpreted by the editor as a command to descend into the correspondingly numbered element of the current expression. Thus:

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

A negative integer has a similar effect, but counting begins from the end of the current expression and proceeds backward, i.e., -1 refers to the last element in the current expression, -2 the next to the last, etc. For either positive integer or negative integer, if there is no such element, an error occurs,† the editor types the faulty command followed by a ?, and then another *. *The current expression is never changed when a command causes an error.* Thus:

†'Editor errors' are not of the flavor described in Chapter 16, i.e., they never cause breaks or even go through the error machinery but are direct calls to error! (p. 16.13) indicating that a command is in some way faulty. What happens next depends on the context in which the command was being executed. For example, there are conditional commands which branch on errors. In most situations, though, an error will cause the editor to type the faulty command followed by a ? and wait for more input. Note that typing control-E while a command is being executed aborts the command exactly as though it had caused an error.

```
*P
(X)
*2

2  ?
*1
*P
X
*
```

A phrase of the form 'the current expression is changed' or 'the current expression becomes' refers to a shift in the editor's _*attention,*_ *not to a modification of the structure being edited.*

When the user changes the current expression by descending into it, the old current expression is not lost. Instead, the editor actually operates by maintaining a *chain* of expressions leading to the current one. The current expression is simply the last link in the chain. Descending adds the indicated subexpression onto the end of the chain, thereby making it be the current expression. The command Ø is used to ascend the chain; it removes the last link of the chain, thereby making the *previous* link be the current expression. Thus:

```
*P
X
*Ø P
(X)
*Ø -1 P
(COND (A Z) (T &))
*
```

Note the use of several commands on a single line in the previous output. The editor operates in a line buffered mode, the same as evalqt. Thus no command is actually seen by the editor, or executed, until the line is terminated, either by a carriage return, or a matching right parenthesis. The user can thus use control-A and control-Q for line-editing edit commands, the same as he does for inputs to evalqt.

In our editing session, we will make the following corrections to append: delete Y from where it appears, add Y to the end of the argument list,† change NUL to NULL, change Z to Y, add Z after CAR, and insert a right parenthesis following CDR X.

First we will delete Y. By now we have forgotten where we are in the function definition, but we want to be at the "top," so we use the command ↑, which ascends through the entire chain of expressions to the top level expression, which then becomes the current expression, i.e., ↑ removes all links except the first one.

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

Note that if we are already at the top, ↑ has no effect, i.e., it is a NOP. However, Ø would generate an error. In other words, ↑ means "go to the top," while Ø means "ascend one link."

† These two operations could be thought of as one operation, i.e., MOVE Y from its current position to a new position, and in fact there is a MOVE command in the editor. However, for the purposes of this introduction, we will confine ourselves to the simpler edit commands.

The basic structure modification commands in the editor are

(n)	$n \geq 1$ deletes the corresponding element from the current expression.
$(n\ e_1, \ldots, e_m)$	$n,m \geq 1$ replaces the <u>nth</u> element in the current expression with $e_1, \ldots, e_m$.
$(-n\ e_1, \ldots, e_m)$	$n,m \geq 1$ inserts $e_1, \ldots, e_m$ before the <u>nth</u> element in the current expression.

Thus:

```
*P
(LAMBDA (X) Y (COND & &))
*(3)
*(2 (X Y))
*P
(LAMBDA (X Y) (COND & &))
*
```

All structure modification done by the editor is destructive, i.e., the editor uses <u>rplaca</u> and <u>rplacd</u> to physically change the structure it was given.

Note that all three of the above commands perform their operation with respect to the <u>nth</u> element from the front of the current expression; the sign of <u>n</u> is used to specify whether the operation is replacement or insertion. Thus, there is no way to specify deletion or replacement of the <u>nth</u> element from the end of the current expression, or insertion before the <u>nth</u> element from the end without counting out that element's position from the front of the list. Similarly, because we cannot specify insertion after a particular element, we cannot attach something at the end of the current expression using the above commands. Instead, we use the command N (for <u>nconc</u>). Thus we could have performed the above changes instead by:

```
*P
(LAMBDA (X) Y (COND & &))
*(3)
*2 (N Y)
*P
(X Y)
*↑ P
*(LAMBDA (X Y) (COND & &))
*
```

Now we are ready to change NUL to NULL. Rather than specify the sequence of descent commands necessary to reach NUL, and then replace it with NULL, i.e., 3 2 1 (1 NULL), we will use F, the find command, to find NUL:

```
*P
(LAMBDA (X Y) (COND & &))
*F NUL
*P
(NUL X)
*(1 NULL)
*Ø P
((NULL X) Z)
*
```

Note that F is special in that it corresponds to *two* inputs. In other words, F says to the editor, "treat your *next* command as an expression to be searched for." The search is carried out in printout order in the current expression. If the target expression is not found there, F automatically ascends and searches those portions of the higher expressions that would appear after (in a printout) the current expression. If the search is successful, the new current expression will be the structure where the expression was found,† and the chain will be the same as one resulting from the appropriate sequence of ascent and descent commands. If the search is not successful, an error occurs, and neither the current expression nor the chain is changed:††

---

† If the search is for an atom, e.g., F NUL, the current expression will be the structure containing the atom. If the search is for a list, e.g., F (NUL X), the current expression will be the list itself.

†† F is never a NOP, i.e., if successful, the current expression after the search will never be the same as the current expression before the search. **Thus F expr repeated without intervening** commands that change the edit chain can be used to find successive instances of expr.

```
*P
((NULL X) Z)
*F COND P

COND  ?
*P
*((NULL X) Z)
*
```

Here the search failed to find a cond following the current expression, although of course a cond does appear earlier in the structure. This last example illustrates 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 (and all commands that may have been typed ahead while the editor was computing) are forgotten.†

We could also have used the R command (for replace) to change NUL to NULL. A command of the form (R e_1 e_2) will replace all occurrences of e_1 in the current expression by e_2. There must be at least one such occurrence or the R command will generate an error. Let us use the R command to change all Z's (even though there is only one) in append to Y:

```
*↑ (R Z Y)
*F Z

Z  ?
*PP
  [LAMBDA (X Y)
    (COND
      ((NULL X)
        Y)
      (T (CONS (CAR)
               (APPEND (CDR X Y]
*
```

†i.e. the input buffer is cleared (and saved), see p. 14.17. It can be restored, i.e., the type-ahead recovered, via the command $ (alt-mode), described in section 22.

The next task is to change (CAR) to (CAR X). We could do this by (R (CAR) (CAR X)), or by:

```
*F CAR
*(N X)
*P
(CAR X)
*
```

The expression we now want to change is the next expression after the current expression, i.e., we are currently looking at (CAR X) in (CONS (CAR X) (APPEND (CDR X Y))). We could get to the append expression by typing 0 and then 3 or -1, or we can use the command NX, which does both operations:

```
*P
(CAR X)
*NX P
(APPEND (CDR X Y))
*
```

Finally, to change (APPEND (CDR X Y)) to (APPEND (CDR X) Y), we could perform (2 (CDR X) Y), or (2 (CDR X)) and (N Y), or 2 and (3), deleting the Y, and then 0 (N Y). However, if y were a complex expression we would not want to have to retype it. Instead, we could use a command which effectively inserts and/or removes left and right parentheses. There are six of these commands: BI, BO, LI, LO, RI, and RO, for both in, both out, left in, left out, right in, and right out. Of course, we will always have the same number of left parentheses as right parentheses, because the parentheses are just a notational guide to structure that is provided by our print program.*
Thus, left in, left out, right in, and right out actually do not insert or remove just one parenthesis, but this is very suggestive of what actually happens.

* Herein lies one of the principal advantages of a LISP oriented editor over a text editor: unbalanced parentheses errors are not possible.

In this case, we would like a right parenthesis to appear following X in (CDR X Y). Therefore, we use the command (RI 2 2), which means insert a right parentheses after the second element in the second element (of the current expression):

```
*P
(APPEND (CDR X Y))
*(RI 2 2)
*P
(APPEND (CDR X) Y)
*
```

We have now finished our editing, and can exit from the editor, to test append, or we could test it while still inside of the editor, by using the E command:

```
*E APPEND((A B) (C D E))
(A B C D E)
*
```

The E command causes the next input to be given to evalqt. If there is another input following it, as in the above example, evalqt will apply the first to the second. Otherwise, evalqt evals the first input.

We prettyprint append, and leave the editor.

```
*PP
  [LAMBDA (X Y)
    (COND
      ((NULL X)
        Y)
      (T (CONS (CAR X)
               (APPEND (CDR X) Y]
*OK
APPEND
←
```

Commands for the New User

As mentioned earlier, the BBN-LISP manual is intended primarily as a reference manual, and the remainder of this chapter is organized and presented accordingly. While the commands introduced in the previous scenario constitute a complete set, i.e., the user could perform any and all editing operations using just those commands, there are many situations in which knowing the right command(s) can save the user considerable effort. We include here as part of the introduction a list of those commands which are not only frequently applicable but also easy to use. They are not presented in any particular order, and are all discussed in detail in the reference portion of the chapter.

UNDO	undoes the last modification to the structure being edited, e.g., if the user deletes the wrong element, UNDO will restore it. The availability of UNDO should give the user confidence to experiment with any and all editing commands, no matter how complex, because he can always reverse the effect of the command.
BK	like NX, except makes the expression immediately *before* the current expression become current.
BF	backwards find. Like F, except searches backwards, i.e., in inverse print order.

\	Restores the current expression to the expression before the last "big jump", e.g., a find command, an ↑, or another \. For example, if the user types F COND, and then F CAR, \ would take him back to the COND. Another \ would take him back to the CAR.
\P	like \ except it restores the edit chain to its state as of the last print, either by P, ?, or PP. If the edit chain has not been changed since the last print, \P restores it to its state as of the printing before that one, i.e., two chains are always saved.

Thus if the user types P followed by 3 2 1 P, \P will take him back to the first P, i.e., would be equivalent to 0 0 0. Another \P would then take him back to the second P, i.e., he can use \P to flip back and forth between two current expressions.

&,--

The search expression given to the F or BF command need not be a literal S-expression. Instead, it can be a pattern. The symbol & can be used anywhere within this pattern to match with any single *element* of a list, and -- can be used to match with any *segment* of a list. Thus, in the incorrect definition of <u>append</u> used earlier, F (NUL &) could have been used to find (NUL X), and F (CDR --) or F (CDR & &), but not F (CDR &), to find (CDR X Y).

Note that & and -- can be nested arbitrarily deeply in the pattern. For example, if there are many places where the variable X is set, F SETQ may not find the desired expression, nor may F (SETQ X &). It may be necessary to use F (SETQ X (LIST --)). However, the usual technique in such a case is to pick out a unique atom which occurs prior to the desired expression and perform two F commands. This "homing in" process seems to be more convenient than ultra-precise specification of the pattern.

$ (alt-mode)	Any atom ending in alt-mode in a pattern will match with the first atom or string that contains the same initial characters. For example, F VER$ will find VERYLONGATOM. $ can be nested inside of the pattern, e.g., F (SETQ VER$ (CONS --)).

If the search is successful, the editor will print = followed by the atom which matched with the $-atom, e.g.,

```
*F (SETQ VER$ &)
=VERYLONGATOM
*
```

Frequently the user will want to replace the entire current expression, or insert something before it. In order to do this using a command of the form (n e_1, ..., e_m) or (-n e_1, ..., e_m), the user must be *above* the current expression. In other words, he would have to perform a 0 followed by a command with the appropriate number. However, if he has reached the current expression via an F command, he may not know what that number is. In this case, the user would like a command whose effect would be to modify the edit chain so that the current expression became the first element in a new, higher current expression. Then he could perform the desired operation via (1 e_1, ..., e_m) or (-1 e_1, ..., e_m). UP is provided for this purpose.

UP

after UP operates, the old current expression is the first element of the new current expression. Note that if the current expression happens to be the first element in the next higher expression, then UP is exactly the same as 0. Otherwise, UP modifies the edit chain so that the new current expression is a tail† of the next higher expression:

```
*F APPEND P
(APPEND (CDR X) Y)
*UP P
... (APPEND & Y))
*0 P
(CONS (CAR X) (APPEND & Y))
*
```

The ... is used by the editor to indicate that the current expression is a tail of the next higher expression as opposed to being an element (i.e., a member) of the next higher expression. Note: if the current expression is *already* a tail, UP has no effect.

†Throughout this chapter 'tail' means 'proper tail', see p. 5.12.

(B e_1, ..., e_m)	inserts e_1, ..., e_m before the current expression, i.e., does an UP and then a -1.
(A e_1, ..., e_m)	inserts e_1, ..., e_m after the current expression, i.e., does an UP and then either a (-2 e_1, ..., e_m) or an (N e_1, ..., e_m), if the current expression is the last one in the next higher expression.
(: e_1, ..., e_m)	replaces current expression by e_1, ..., e_m, i.e., does an UP and then a (1 e_1, ..., e_m).
DELETE	deletes current expression, i.e., equivalent to (:).

Earlier, we introduced the RI command in the append example. The rest of the commands in this family: BI, BO, LI, LO, and RO, perform similar functions and are useful in certain situations. In addition, the commands MBD and XTR can be used to combine the effects of several commands of the BI-BO family. MBD is used to embed the current expression in a larger expression. For example, if the current expression is (PRINT bigexpression), and the user wants to replace it by (COND (FLG (PRINT bigexpression))), he can accomplish this by (LI 1), (-1 FLG), (LI 1), and (-1 COND), or by a single MBD command.

XTR is used to extract an expression from the current expression. For example, extracting the PRINT expression from the above COND could be accomplished by (1), (LO 1), (1), and (LO 1) or by a single XTR command. The new user is encouraged to include XTR and MBD in his repertoire as soon as he is familiar with the more basic commands.

Attention Changing Commands

Commands to the editor fall into three classes: commands that change the current expression (i.e., change the edit chain) thereby "shifting the editor's attention," commands that modify the structure being edited, and miscellaneous commands, e.g., exiting from the editor, printing, evaluating expressions.

Within the context of commands that shift the editor's attention, we can distinguish among (1) those commands whose operation depends only on the *structure* of the edit chain, e.g., Ø, UP, NX; (2) those which depend on the *contents* of the structure, i.e., commands that search; and (3) those commands which simply restore the edit chain to some previous state, e.g., \ , \P. (1) and (2) can also be thought of as local, small steps versus open ended, big jumps. Commands of type (1) are discussed on pp. 9.18 - 9.23; type (2) on pp. 9.24 - 9.38; and type (3) on pp. 9.39 - 9.40.

Local Attention-Changing Commands

UP

(1) If a P command would cause the editor to type ... before typing the current expression, i.e. the current expression is a tail of the next higher expression, UP has no effect; otherwise

(2) UP modifies the edit chain so that the old current expression (i.e., the one at the time UP was called) is the first element in the new current expression.†

Examples: The current expression in each case is (COND ((NULL X) (RETURN Y))).

```
1.   *1 P
     COND
     *UP P
     (COND (& &))

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

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

†If the current expression is the first element in the next higher expression UP simply does a Ø. Otherwise UP adds the corresponding tail to the edit chain.

The execution of UP is straightforward, except in those cases where the current expression appears more than once in the next higher expression. For example, if the current expression is (A NIL B NIL C NIL) and the user performs 4 followed by UP, the current expression should then be ... NIL C NIL). UP can determine which tail is the correct one because the commands that descend save the last tail on an internal editor variable, lastail. Thus after the 4 command is executed, lastail is (NIL C NIL). When UP is called, it first determines if the current expression is a tail of the next higher expression. If it is, UP is finished. Otherwise, UP computes memb[current-expression;next-higher-expression] to obtain a tail beginning with the current expression.† If there are no other instances of the current-expression in the next higher expression, this tail is the correct one. Otherwise UP uses lastail to select the correct tail.††

---

†The current expression should *always* be either a tail or an element of the next higher expression. If it is neither, for example the user has directly (and incorrectly) manipulated the edit chain, UP generates an error.

††Occasionally the user can get the edit chain into a state where lastail cannot resolve the ambiguity, for example if there were two non-atomic structures in the same expression that were eq, and the user descended more than one level into one of them and then tried to come back out using UP. In this case, UP selects the first tail and prints LOCATION UNCERTAIN to warn the user. Of course, we could have solved this problem completely in our implementation by saving at each descent *both* elements and tails. However, this would be a costly solution to a situation that arises infrequently, and when it does, has no detrimental effects. The lastail solution is cheap and resolves 99% of the ambiguities.

n (n>0)	adds the _n_th element of the current expression to the front of the edit chain, thereby making it be the new current expression. Sets _lastail_ for use by UP. Generates an error if the current expression is not a list that contains at least _n_ elements.
-n (n>0)	adds the _n_th element from the end of the current expression to the front of the edit chain, thereby making it be the new current expression. Sets _lastail_ for use by UP. Generates an error if the current expression is not a list that contains at least _n_ elements.
Ø	Sets edit chain to _cdr_ of edit chain, thereby making the next higher expression be the new current expression. Generates an error if there is no higher expression, i.e. _cdr_ of edit chain is NIL.

Note that Ø usually corresponds to going back to the next higher left parenthesis, but not always. For example, if the current expression is (A B C D E F G), and the user performs

```
*3 UP P
... C D E F G)
*3 UP P
... E F G)
*Ø P
... C D E F G)
```

If the intention is to go back to the next higher left parenthesis, regardless of any intervening tails, the command !Ø can be used.†

†!Ø is pronounced bang-zero.

!Ø	does repeated Ø's until it reaches a point where the current expression is *not* a tail of the next higher expression, i.e., always goes back to the next higher left parenthesis.
↑	sets edit chain to last of edit chain, thereby making the top level expression be the current expression. Never generates an error.
NX	effectively does an UP followed by a 2,[†] thereby making the current expression be the next expression. Generates an error if the current expression is the last one in a list. (However, !NX described below will handle this case.)
BK	makes the current expression be the previous expression in the next higher expression. Generates an error if the current expression is the first expression in a list.

For example, if the current expression is (COND ((NULL X) (RETURN Y)))

```
*F RETURN P
(RETURN Y)
*BK P
(NULL X)
```

[†] Both NX and BK operate by performing a !Ø followed by an appropriate number, i.e. there won't be an extra tail above the new current expression, as there would be if NX operated by performing an UP followed by a 2.

(NX n) n>0	equivalent to n NX commands, except if an error occurs, the edit chain is not changed.
(BK n) n>0	equivalent to n BK commands, except if an error occurs, the edit chain is not changed

Note: (NX -n) is equivalent to (BK n), and vice versa.

!NX	makes current expression be the next expression at a higher level, i.e., goes through any number of right parentheses to get to the next expression.

For example:

```
*PP
  (PROG ((L L)
         (UF L))
    LP   (COND
           ((NULL (SETQ L (CDR L)))
              (ERROR!))
           ([NULL (CDR (FMEMB (CAR L)
                                   (CADR L]

             (GO LP)))
         (EDITCOM (QUOTE NX))
         (SETQ UNFIND UF)
         (RETURN L))
*F CDR P
(CDR L)
*NX

NX   ?
*!NX P
(ERROR!)
*!NX P
((NULL &) (GO LP))
*!NX P
(EDITCOM (QUOTE NX))
*
```

!NX operates by doing ∅'s until it reaches a stage where the current expression is *not* the last expression in the next higher expression, and then does a NX. Thus !NX always goes through at least one unmatched right parenthesis, and the new current expression is always on a different level, i.e. !NX and NX always produce different results. For example using the previous current expression:

```
*F CAR P
(CAR L)
*!NX P
(GO LP)
*\P P
(CAR L)
*NX P
(CADR L)
*
```

(NTH n) n≠0	equivalent to n followed by UP, i.e., causes the list starting with the nth element of the current expression (or nth from the end if n<0) to become the current expression.* Causes an error if current expression does not have at least n elements.

A generalized form of NTH using location specifications is described on page 9.37.

*(NTH 1) is a NOP.

Commands That Search

All of the editor commands that search use the same pattern matching routine.† We will therefore begin our discussion of searching by describing the pattern match mechanism. A pattern pat matches with x if

1. pat is eq to x.
2. pat is &.
3. pat is a number and eqp to x.
4. pat is a string and strequal[pat;x] is true.
5. If car[pat] is the atom *ANY*, cdr[pat] is a list of patterns, and pat matches x if and only if one of the patterns on cdr[pat] matches x.
6. If pat is a literal atom or string, and nthchar[pat;-1] is $ (alt-mode), then pat matches with any literal atom or string which has the same initial characters as pat, e.g. VER$ matches with VERYLONGATOM, as well as "VERYLONGSTRING".

† This routine is available to the user directly, and is described later in this chapter in the section on "Editor Functions."

7. If car[pat] is the atom --, pat matches x if
   a. cdr[pat]=NIL, i.e. pat=(--), e.g.
      (A --) matches (A) (A B C) and (A . B)
      In other words, -- can match any tail of a list.
   b. cdr[pat] matches with some tail of x,
      e.g. (A -- (&)) will match with (A B C (D)),
      but not (A B C D), or (A B C (D) E). However,
      note that (A -- (&) --) will match with
      (A B C (D) E).
      In other words, -- can match any interior
      segment of a list.

8. If car[pat] is the atom ==, pat matches x if and only if cdr[pat] is eq to x.[†]

9. Otherwise if x is a list, pat matches x if car[pat] matches car[x], and cdr[pat] matches cdr[x].

When searching, the pattern matching routine is called only match with *elements* in the structure, unless the pattern begins with ..., in which case cdr of the pattern is matched against tails in the structure.[††] Thus if the current expression is (A B C (B C)),

```
*F (B --)
*P
(B C)
*0 F (... B --)
*P
... B C (B C))
*F (... B --)
*P
(B C)
*
```

[†]Pattern 8 is for use by programs that call the editor as a subroutine, since any non-atomic expression in a command *typed in* by the user obviously cannot be *eq* to existing structure.

[††]In this case, the tail does not have to be a proper tail, e.g. (... A --) will match with the element (A B C) as well as with cdr of (X A B C), since (A B C) is a tail of (A B C).

Search Algorithm

Searching begins with the current expression and proceeds in print order. Searching usually means find the *next* instance of this pattern, and consequently a match is not attempted that would leave the edit chain unchanged.* At each step, the pattern is matched against the next element in the expression currently being searched, unless the pattern begins with ... in which case it is matched against the corresponding tail of the expression.**

If the match is not successful, the search operation is recursive first in the car direction and then in the cdr direction, i.e., if the element under examination is a list, the search descends into that list before attempting to match with other elements (or tails) at the same level.***

* However, there is a version of the find command which can succeed and leave the current expression unchanged.

** eq[pattern;tail-of-expression]=T also indicates a successful match, so that a search for FOO will find the FOO in (FIE . FOO). The only exception to this occurs when pattern=NIL, e.g., F NIL. In this case, the pattern will not match with a null tail (since most lists end in NIL) but will match with a NIL element.

***There is also a version of the find command which only attempts matches at the top level of the current expression, i.e., does not descend into elements, or ascend to higher expressions.

However, at no point is the total recursive depth of the search (sum of number of cars and cdrs descended into) allowed to exceed the value of the variable maxlevel. At that point, the search of that element or tail is abandoned, exactly as though the element or tail had been completely searched without finding a match, and the search continues with the next element or tail for which the recursive depth is below maxlevel. This feature is designed to enable the user to search circular list structures (by setting maxlevel small), as well as protecting him from accidentally encountering a circular list structure in the course of normal editing. maxlevel is initially set to 300.*

If a successful match is not found in the current expression, the search automatically ascends to the next higher expression,** and continues searching there on the next expression after the expression it just finished searching. If there is none, it ascends again, etc. This process continues until the entire edit chain has been searched, at which point the search fails, and an error is generated. If the search fails (or, what is equivalent, is aborted by Control-E), the edit chain is not changed (nor are any conses performed).

If the search is successful, i.e., an expression is found that the pattern matches, the edit chain is set to the value it would have had had the user reached that expression via a sequence of integer commands.

*maxlevel is a globalvar (see p. 18.6). If changed, it must be *reset* not rebound.

See footnote * on previous page.

If the expression that matched was a list, it will be the final link in the edit chain, i.e., the new current expression. If the expression that matched is not a list, e.g., is an atom, the current expression will be the tail beginning with that atom,* i.e., that atom will be the first element in the new current expression. In other words, the search effectively does an UP.**

---

*Except for situations where match is with _y_ in (x . y), _y_ atomic and not NIL. In this case, the current expression will be (x . y).

** Unless _upfindflg_=NIL (initially set to T). For discussion, see pp. 9.49-9.50.

Search Commands

All of the commands below set lastail for use by UP, set unfind for use by \ (p. 9.39), and do not change the edit chain or perform any conses if they are unsuccessful or aborted.

F pattern	i.e., two commands: the F informs the editor that the *next* command is to be interpreted as a pattern. This is the most common and useful form of the find command. If successful, the edit chain always changes, i.e., F pattern means find the next instance of pattern.
	If memb[pattern;current-expression] is true, F does not proceed with a full recursive search.
	If the value of the memb is NIL, F invokes the search algorithm described earlier.

Thus if the current expression were (PROG NIL LP (COND (-- (GO LP1))) ... LP1 ...), F LP1 would find the prog label, not the LP1 inside of the GO expression, even though the latter appears first (in print order) in the current expression. Note that 1 (making the atom PROG be the current expression), followed by F LP1 *would* find the first LP1.

(F pattern N)	same as F pattern, i.e., finds the next instance of pattern, except the *memb* check of F pattern is not performed.
(F pattern T)	Similar to F pattern, except may succeed without changing edit chain, and does not perform the *memb* check.

Thus if the current expression is (COND ..), F COND will look for the next COND, but (F COND T) will 'stay here'.

(F pattern n) n>0	Finds the *n*th place that pattern matches. Equivalent to (F pattern T) followed by (F pattern N) repeated n-1 times. Each time *pattern* successfully matches, *n* is decremented by 1, and the search continues, until n reaches 0. Note that the pattern does not have to match with *n* identical expressions; it just has to match *n* times. Thus if the current expression is (FOO1 FOO2 FOO3), (F FOO$ 3) will find FOO3. If the pattern does not match successfully *n* times, an error is generated and the edit chain is unchanged (even if the *pattern* matched n-1 times).
(F pattern) or (F pattern NIL)	only matches with elements at the top level of the current expression, i.e., the search will not descend into the current expression, nor will it go outside of the current expression. May succeed without changing edit chain.

For example, if the current expression is
(PROG NIL (SETQ X (COND & &)) (COND &) ...)
F (COND --) will find the COND inside the SETQ, whereas
(F (COND --)) will find the top level COND, i.e., the second one.

(FS pattern$_1$ ... pattern$_n$)   equivalent to F pattern$_1$ followed by F pattern$_2$ ... followed by F pattern$_n$, so that if F pattern$_m$ fails, edit chain is left at place pattern$_{m-1}$ matched.

(F= expression x)   equivalent to (F (== . expression) x), i.e., searches for a structure eq to expression, see p. 9.25.

(ORF pattern$_1$ ... pattern$_n$)   equivalent to (F (*ANY* pattern$_1$ ... pattern$_n$) N), i.e., searches for an expression that is matched by either pattern$_1$ or ... pattern$_n$. See p. 9.24.

BF pattern   backwards find. Searches in reverse print order, beginning with expression immediately before the current expression (unless the current expression is the top level expression, in which case BF searches the entire expression, in reverse order).

BF uses the same pattern match routine as F, and maxlevel and upfindflg have the same effect, but the searching begins at the *end* of each list, and descends into each element before attempting to match that element. If unsuccessful, the search continues with the next previous element, etc., until the front of the list is reached, at which point BF ascends and backs up, etc.

For example, if the current expression is
(PROG NIL (SETQ X (SETQ Y (LIST Z))) (COND ((SETQ W --) --)) --)
F LIST followed by BF SETQ will leave the current expression as
(SETQ Y (LIST Z)), as will F COND followed by BF SETQ.

(BF pattern T)	search always includes current expression, i.e., starts at end of current expression and works backward, then ascends and backs up, etc.

Thus in the previous example, where F COND followed by BF SETQ found (SETQ Y (LIST Z)), F COND followed by (BF SETQ T) would find the (SETQ W --) expression.

(BF pattern) (BF pattern NIL)	same as BF pattern.

Location Specification

Many of the more sophisticated commands described later in this chapter use a more general method of specifying position called a location specification. A location specification is a list of edit commands that are executed in the normal fashion with two exceptions. First, all commands not recognized by the editor are interpreted as though they had been preceded by F.* For example, the location specification (COND 2 3) specifies the 3rd element in the first clause of the next COND.**

Secondly, if an error occurs while evaluating one of the commands in the location specification, and the edit chain had been changed, i.e., was not the same as it was at the beginning of that execution of the location specification, the location operation will continue. In other words, the location operation keeps going unless it reaches a state where it detects that it is 'looping', at which point it gives up. Thus, if (COND 2 3) is being located, and the first clause of the next COND contained only two elements, the execution of the command 3 would cause an error. The search would then continue by looking for the next COND. However, if a point were reached where there were no further CONDs, then the first command, COND, would cause the error; the edit chain would not have been changed, and so the entire location operation would fail, and cause an error.

*Normally such commands would cause errors.

**Note that the user could always write (F COND 2 3) for (COND 2 3) if he were not sure whether or not COND was the name of an atomic command.

The IF command and the ## function provide a way of using in location specifications arbitrary predicates applied to elements in the current expression. IF and ## will be described in detail later in the chapter, along with examples illustrating their use in location specifications.

Throughout this chapter, the meta-symbol @ is used to denote a location specification. Thus @ is a list of commands interpreted as described above. @ can also be atomic, in which case it is interpreted as list[@].

(LC . @)	provides a way of explicitly invoking the location operation, e.g. (LC COND 2 3) will perform the search described above.
(LCL . @)	Same as LC except search is confined to current expression, i.e., the edit chain is rebound during the search so it looks as if the editor were called on just the current expression. For example, to find a COND containing a RETURN, one might use the location specification (COND (LCL RETURN) \\) where the \\ would reverse the effects of the LCL command, and make the final current expression be the COND.
(2ND . @)	Same as (LC . @) followed by another (LC . @) except that if the first succeeds and second fails, no change is made to the edit chain.
(3RD . @)	Similar to 2ND.

(← pattern) ascends the edit chain looking for a link which matches pattern. In other words, it keeps doing 0's until it gets to a specified point. If pattern is atomic, it is matched with the first element of each link, otherwise with the entire link.[†]

For example:

```
*PP
  [PROG NIL
        (COND
          [(NULL (SETQ L (CDR L)))
            (COND
              (FLG (RETURN L]
          ([NULL (CDR (FMEMB (CAR L)
                                   (CADR L]
            (GO LP]
*F CADR
*(← COND)
*P
(COND (& &) (& &))
*
```

Note that this command differs from BF in that it does not search *inside* of each link, it simply ascends. Thus in the above example, F CADR followed by BF COND would find (COND (FLG (RETURN L))), not the higher COND.

If no match is found, an error is generated and the edit chain is unchanged.

[†] If pattern is of the form (IF expression), expression is evaluated at each link, and if its value is NIL, or the evaluation causes an error, the ascent continues.

(BELOW com x)	ascends the edit chain looking for a link specified by com, and stops x* links below that, i.e. BELOW keeps doing Ø's until it gets to a specified point, and then backs off n Ø's.
(BELOW com)	same as (BELOW com 1).

For example, (BELOW COND) will cause the cond *clause* containing the current expression to become the new current expression. Thus if the current expression is as show above, F CADR followed by (BELOW COND) will make the new expression be ([NULL (CDR (FMEMB (CAR L) CADR L] (GO LP)), and is therefore equivalent to Ø Ø Ø Ø.

BELOW operates by evaluating x and then executing com, or (← com) if com is not a recognized edit command, and measuring the length of the edit chain at that point. If that length is m and the length of the current edit chain is n, then BELOW ascends n-m-y links where y is the value of x. Generates an error if com causes an error, i.e., it can't find the higher link, or if n-m-y is negative.

The BELOW command is useful for locating a substructure by specifying something it contains. For example, suppose the user is editing a list of lists, and wants to find a sublist that contains a FOO (at any depth). He simply executes F FOO (BELOW \).

* x is evaluated, e.g., (BELOW com (IPLUS X Y)).

(NEX x) same as (BELOW x) followed by NX.

For example, if the user is deep inside of a SELECTQ clause, he can advance to the next clause with (NEX SELECTQ).

NEX same as (NEX ←).

The atomic form of NEX is useful if the user will be performing repeated executions of (NEX x). By simply MARKing (see p. 9.39) the chain corresponding to x, he can use NEX to step through the sublists.

(NTH @) generalized NTH command. Effectively performs (LCL . @), followed by (BELOW \), followed by UP.

In other words, NTH locates @, using a search restricted to the current expression, and then backs up to the current level, where the new current expression is the tail whose first element contains, however deeply, the expression that was the terminus of the location operation. For example:

```
*P
(PROG (& &) LP (COND & &) (EDITCOM &) (SETQ UNFIND UF) (RETURN L))
*(NTH UF)
*P
... (SETQ UNFIND UF) (RETURN L))
*
```

If the search is unsuccessful, NTH generates an error and the edit chain is not changed.

Note that (NTH n) is just a special case of (NTH @), and in fact, no special check is made for @ a number; both commands are executed identically.

(pattern .. . @)[†] e.g., (COND .. RETURN). Finds a cond that contains a return, at any depth. Equivalent to (F pattern N), (LCL . @) followed by (← pattern).

For example, if the current expression is (PROG NIL (COND ((NULL L) (COND (FLG (RETURN L))))) --), then (COND .. RETURN) will make (COND (FLG (RETURN L))) be the current expression. Note that it is the innermost COND that is found, because this is the first COND encountered when ascending from the RETURN. In other words, (pattern .. @) is *not* equivalent to (F pattern N), followed by (LCL . @) followed by \.

Note that @ is a location specification, not just a pattern. Thus (RETURN .. COND 2 3) can be used to find the RETURN which contains a COND whose first clause contains (at least) three elements. Note also that since @ permits any edit command, the user can write commands of the form (COND .. (RETURN .. COND)), which will locate the first COND that contains a RETURN that contains a COND.

[†]An infix command, '..' is not a meta-symbol, it *is* the name of the command. @ is cddr of the command.

Commands That Save and Restore The Edit Chain

Three facilities are available for saving the current edit chain and later retrieving it. The commands are MARK, which marks the current chain for future reference, ←,[†] which returns to the last mark without destroying it, and ←←, which returns to the last mark and also erases it.

MARK	adds the current edit chain to the front of the list _marklst_.
←	makes the new edit chain be (CAR MARKLST). Generates an error if _marklst_ is NIL, i.e., no MARKS have been performed, or all have been erased.
←←	similar to ← but also erases the MARK, i.e., performs (SETQ MARKLST (CDR MARKLST)).

If the user did not prepare in advance for returning to a particular edit chain, he may still be able to return to that chain with a single command by using \ or \P.

\	makes the edit chain be the value of _unfind_. Generates an error if _unfind_=NIL.

[†]An atomic command; do not confuse with the list command (← pattern).

unfind is set to the current edit chain by each command that makes a "big jump", i.e., a command that usually performs more than a single ascent or descent, namely ↑, ←, ←←, !NX, all commands that involve a search, e.g., F, LC, .., BELOW, et al and \ and \P themselves.*

For example, if the user types F COND, and then F CAR, \ would take him back to the COND. Another \ would take him back to the CAR, etc.

\P	restores the edit chain to its state as of the last print operation, i.e. P, ?, or PP. If the edit chain has not changed since the last printing, \P restores it to its state as of the printing before that one, i.e., two chains are always saved.

For example, if the user types P followed by 3 2 1 P, \P will return to the first P, i.e., would be equivalent to 0 0 0.** Another \P would then take him back to the second P, i.e., the user could use \P to flip back and forth between the two edit chains.

(S var . @)	sets var (using setq) to the current expression after performing (LC . @). Edit chain is not changed.

Thus (S FOO) will set foo to the current expression, (S FOO -1 1) will set foo to the first element in the last element of the current expression.

*Except that unfind is not reset when the current edit chain is the top level expression, since this could always be returned to via the ↑ command.

**Note that if the user had typed P followed by F COND, he could use *either* \ or \P to return to the P, i.e., the action of \ and \P are independent.

Commands That Modify Structure

The basic structure modifications commands in the editor are:

(n)	$n \geq 1$ deletes the corresponding element from the current expression.
(n e_1 ... e_m)	$n,m \geq 1$ replaces the nth element in the current expression with e_1 ... e_m.
(-n e_1 ... e_m)	$n,m \geq 1$ inserts e_1 ... e_m before the nth element in the current expression.
(N e_1 ... e_m)	$m \geq 1$ attaches e_1 ... e_m at the end of the current expression.

As mentioned earlier:

all structure modification done by the editor is destructive, i.e. the editor uses rplaca and rplacd to physically change the structure it was given.

However, all structure modification is undoable, see UNDO p. 9.83.

All of the above commands generate errors if the current expression is not a list, or in the case of the first three commands, if the list contains fewer than n elements. In addition, the command (1), i.e. delete the first element, will cause an error if there is only one element, since deleting the first element must be done by replacing it with the second element, and then deleting the second element. Or, to look at it another way, deleting the first element when there is only one element would require changing a list to an atom (i.e. to NIL) which cannot be done.*

*However, the command DELETE will work even if there is only one element in the current expression, since it will ascend to a point where it *can* do the deletion.

Implementation of Structure Modification Commands

Note: Since all commands that insert, replace, delete or attach structure use the same low level editor functions, the remarks made here are valid for all structure changing commands.

For all replacement, insertion, and attaching at the end of a list, unless the command was typed in directly to the editor,* *copies* of the corresponding structure are used, because of the possibility that the exact same command, (i.e. same list structure) might be used again. Thus if the program constructs the command (1 (A B C)) via (LIST 1 FOO), and gives this command to the editor, the (A B C) used for the replacement will *not* be eq to foo.**

The rest of this section is included for applications wherein the editor is used to modify a data structure, and pointers into that data structure are stored elsewhere. In these cases, the actual mechanics of structure modification must be known in order to predict the effect that various commands may have on these outside pointers. For example, if the value of FOO is cdr of the current expression, what will the commands (2), (3), (2 X Y Z), (-2 X Y Z), etc. do to foo?

---

*Some editor commands take as arguments a list of edit commands, e.g. (LP F FOO (1 (CAR FOO))). In this case, the command (1 (CAR FOO)) is not considered to have been "typed in" even though the LP command itself may have been typed in. Similarly, commands originating from macros, or commands given to the editor as arguments to editf, editv, et al, e.g. EDITF(FOO F COND (N --)) are not considered typed in.

**The user can circumvent this by using the I command, which computes the structure to be used. In the above example, the form of the command would be (I 1 FOO), which would replace the first element with the value of foo itself. See p. 9.70.

Deletion of the first element in the current expression is performed by replacing it with the second element and deleting the second element by patching around it. Deletion of any other element is done by patching around it, i.e., the previous tail is altered. Thus if foo is eq to the current expression which is (A B C D), and fie is cdr or foo, after executing the command (1), foo will be (B C D) (which is equal but not eq to fie). However, under the same initial conditions, after executing (2) fie will be unchanged, i.e., fie will still be (B C D) even though the current expression and foo are now (A C D).*

Both replacement and insertion are accomplished by smashing both car and cdr of the corresponding tail. Thus, if foo were eq to the current expression, (A B C D), after (1 X Y Z), foo would be (X Y Z B C D). Similarly, if foo were eq to the current expression, (A B C D), then after (-1 X Y Z), foo would be (X Y Z A B C D).

The N command is accomplished by smashing the last cdr of the current expression a la nconc. Thus if foo were eq to any tail of the current expression, after executing an N command, the corresponding expressions would also appear at the end of foo.

In summary, the only situation in which an edit operation will *not* change an external pointer occurs when the external pointer is to a *proper tail* of the data structure, i.e., to cdr of some node in the structure, and the operation is deletion. If all external pointers are to *elements* of the structure, i.e., to car of some node, or if only insertions, replacements, or attachments are performed, the edit operation will *always* have the same effect on an external pointer as it does on the current expression.

* A general solution of the problem just isn't possible, as it would require being able to make two lists eq to each other that were originally different. Thus if fie is cdr of the current expression, and fum is cddr of the current expression, performing (2) would have to make fie be eq to fum if all subsequent operations were to update both fie and fum correctly. Think about it.

The A,B,: Commands

In the (n), (n $e_1, \ldots, e_m$), and (-n $e_1, \ldots, e_m$) commands, the sign of the integer is used to indicate the operation. As a result, there is no direct way to express insertion after a particular element, (hence the necessity for a separate N command). Similarly, the user cannot specify deletion or replacement of the nth element from the end of a list without first converting n to the corresponding positive integer. Accordingly, we have:

(B $e_1, \ldots, e_m$)	inserts $e_1, \ldots, e_m$ before the current expression. Equivalent to UP followed by (-1 $e_1, \ldots, e_m$).

For example, to insert FOO before the last element in the current expression, perform -1 and then (B FOO).

(A $e_1, \ldots, e_m$)	inserts $e_1, \ldots, e_m$ after the current expression. Equivalent to UP followed by (-2 $e_1, \ldots, e_m$) or (N $e_1, \ldots, e_m$) whichever is appropriate.
(: $e_1, \ldots, e_m$)	replaces the current expression by $e_1, \ldots, e_m$. Equivalent to UP followed by (1 $e_1, \ldots, e_m$).

DELETE or (:) deletes the current expression, or if the current expression is a tail, deletes its first element.

DELETE first tries to delete the current expression by performing an UP and then a (1). This works in most cases. However, if after performing UP, the new current expression contains only one element, the command (1) will not work. Therefore, DELETE starts over and performs a BK, followed by UP, followed by (2). For example, if the current expression is (COND ((MEMB X Y)) **(T Y))**, and the user performs -1, and then DELETE, the BK-UP-(2) method is used, and the new current expression will be ... **((MEMB X Y)))**

However, if the next higher expression contains only one element, BK will not work. So in this case, DELETE performs UP, followed by (: NIL), i.e., it *replaces* the higher expression by NIL. For example, if the current expression is (COND ((MEMB X Y)) (T Y)) and the user performs F MEMB and then DELETE, the new current expression will be ... NIL **(T Y))** and the original expression would now be (COND NIL **(T Y))**. The rationale behind this is that deleting (MEMB X Y) from ((MEMB X Y)) changes a list of one element to a list of no elements, i.e., () or NIL. Note that 2 followed by DELETE would *delete* ((MEMB X Y)) *not* replace it by NIL.

If the current expression is a tail, then B, A, and : will work exactly the same as though the current expression were the first element in that tail. Thus if the current expression were ... (PRINT Y) (PRINT Z)), (B (PRINT X)) would insert (PRINT X) before (PRINT Y), leaving the current expression ... (PRINT X) (PRINT Y) (PRINT Z)).

The following forms of the A, B, and : commands incorporate a location specification:

(INSERT $e_1, \ldots, e_m$ BEFORE . @)* Similar to (LC . @) followed by (B $e_1, \ldots, e_m$).

```
*P
(PROG (& & X) **COMMENT** (SELECTQ ATM & NIL) (OR & &) (PRIN1 & T)
(PRIN1 & T) (SETQ X &   †

*(INSERT LABEL BEFORE PRIN1)
*P
(PROG (& & X) **COMMENT** (SELECTQ ATM & NIL) (OR & &) LABEL (PRIN1 &
T) (
*
```

Current edit chain is not changed, but <u>unfind</u> is set to the edit chain after the B was performed, i.e. \ will make the edit chain be that chain where the insertion was performed.

(INSERT $e_1, \ldots, e_m$ AFTER . @) similar to INSERT BEFORE except uses A instead of B.

(INSERT $e_1, \ldots, e_m$ FOR . @) similar to INSERT BEFORE except uses : for B.

*i.e. @ is cdr[member[BEFORE;command]]

†Sudden termination of output followed by an extra carriage return indicates printing was aborted by control-E.

(REPLACE @ WITH $e_1, \ldots, e_m$)† Here @ is the *segment* of the command between REPLACE and WITH. Same as (INSERT $e_1, \ldots, e_m$ FOR . @).

Example: (REPLACE COND -1 WITH (T (RETURN L)))

(CHANGE @ TO $e_1, \ldots, e_m$) Same as REPLACE WITH

(DELETE . @) does a (LC . @) followed by DELETE. Current edit chain is not changed,* but unfind is set to the edit chain after the DELETE was performed.

Example: (DELETE -1), (DELETE COND 3)

Note that if @ is NIL (empty), the corresponding operation is performed *here* (on the current edit chain), e.g. (REPLACE WITH (CAR X)) is equivalent to (: (CAR X)). For added readability, HERE is also permitted, e.g. (INSERT (PRINT X) BEFORE HERE) will insert (PRINT X) before the current expression (but not change the edit chain).

Note also that @ does not have to specify a location *within* the current expression, i.e. it is perfectly legal to ascend to INSERT, REPLACE, or DELETE. For example (INSERT (RETURN) AFTER ↑ PROG -1) will go to the top, find the first prog, and insert a (RETURN) at its end, and not change the current edit chain.

†BY can be used for WITH.

*Unless the current expression is no longer a part of the expression being edited, e.g. if the current expression is ... C) and the user performs (DELETE 1), the tail, (C), will have been cut off. Similarly, if the current expression is (CDR Y) and the user performs (REPLACE WITH (CAR X)).

Finally, the A, B, and : commands, (and consequently INSERT, REPLACE, and CHANGE), all make special checks in e_1 thru e_m for expressions of the form (## . coms). In this case, the expression used for inserting or replacing is a *copy* of the current expression after executing coms, a list of edit commands.* For example, (INSERT (## F COND -1 -1) AFTER 3)** will make a copy of the last form in the last clause of the next cond, and insert it after the third element of the current expression.

*The execution of coms does not change the current edit chain.

***Not* (INSERT F COND -1 (## -1) AFTER 3), which inserts four elements after the third element, namely F, COND, -1, and a copy of the last element in the current expression.

Form Oriented Editing and the Role of UP

The UP that is performed before A, B, and : commands,* makes these operations form-oriented. For example, if the user types F SETQ, and then DELETE, or simply (DELETE SETQ), he will delete the entire SETQ expression, whereas (DELETE X) if X is a variable, deletes just the variable X. In both cases, the operation is performed on the corresponding *form* and in both cases is probably what the user intended. Similarly, if the user types (INSERT (RETURN Y) BEFORE SETQ), he means before the SETQ expression, not before the atom SETQ.** A consequent of this procedure is that a pattern of the form (SETQ Y --) can be viewed as simply an elaboration and further refinement of the pattern SETQ. Thus (INSERT (RETURN Y) BEFORE SETQ) and (INSERT (RETURN Y) BEFORE (SETQ Y --)) perform the same operation*** and, in fact, this is one of the motivations behind making the current expression after F SETQ, and F (SETQ Y --) be the same.

Occasionally, however, a user may have a data structure in which no special significance or meaning is attached to the position of an atom in a list, as LISP attaches to atoms that appear as car of a list, versus those appearing elsewhere in a list. In general, the user may not even *know* whether a particular atom is at the head of a list or not. Thus, when he writes (INSERT expression AFTER FOO), he means after the atom FOO, whether

---

*and therefore in INSERT, CHANGE, REPLACE, and DELETE commands after the location portion of the operation has been performed.

**There is some ambiguity in (INSERT expr AFTER functionname), as the user might mean make expr be the function's first argument. Similarly, the user cannot write (REPLACE SETQ WITH SETQQ) meaning change the name of the function. The user must in these cases write (INSERT expr AFTER functionname 1), and (REPLACE SETQ 1 WITH SETQQ).

***assuming the next SETQ is of the form (SETQ Y -)).

or not it is car of a list. By setting the variable upfindflg to NIL[†] the user can suppress the implicit UP that follows searches for atoms, and thus achieve the desired effect. With upfindflg=NIL then following F FOO, for example, the current expression will be the atom FOO. In this case, the A, B, and : operations will operate with respect to the atom FOO. If the user intends the operation to refer to the list which FOO heads, he simply uses instead the pattern (FOO --).

[†]Initially, and usually, set to T.

Extract and Embed

Extraction involves replacing the current expression with one of its subexpressions (from any depth).

(XTR . @)	replaces the original current expression with the expression that is current after performing (LCL . @).

For example, if the current expression is (COND ((NULL X) (PRINT Y))), (XTR PRINT), or (XTR 2 2) will replace the cond by the print.

If the current expression after (LCL . @) is a *tail* of a higher expression, its first element is used.

For example, if the current expression is (COND ((NULL X) Y) (T Z)), then (XTR Y) will replace the cond with Y.

If the extracted expression is a list, then after XTR has finished, the current expression will be that list.

Thus, in the first example, the current expression after the XTR would be (PRINT Y).

If the extracted expression is not a list, the new current expression will be a tail whose first element is that non-list.

Thus, in the second example, the current expression after the XTR would be ... Y followed by whatever followed the COND.

If the current expression *initially* is a tail, extraction works exactly the same as though the current expression were the first element in that tail. Thus if the current expression is ... (COND ((NULL X) (PRINT Y))) (RETURN Z)), then (XTR PRINT) will replace the <u>cond</u> by the <u>print</u>, leaving (PRINT Y) as the current expression.

The extract command can also incorporate a location specification.

(EXTRACT $@_1$ FROM . $@_2$)[†]	Performs (LC . $@_2$) and then (XTR . $@_1$). Current edit chain is not changed, but <u>unfind</u> is set to the edit chain after the XTR was performed.

Example: If the current expression is (PRINT (COND ((NULL X) Y) (T Z))) then following (EXTRACT Y FROM COND), the current expression will be (PRINT Y). (EXTRACT 2 -1 FROM COND), (EXTRACT Y FROM 2), (EXTRACT 2 -1 FROM 2) will all produce the same result.

[†] $@_1$ is the *segment* between EXTRACT and FROM.

While extracting replaces the current expression by a subexpression, embedding replaces the current expression with one containing *it* as a subexpression.

(MBD x)	x is a list, substitutes[†] the current expression for all instances of the atom * in x, and replaces the current expression with the result of that substitution.

Example: If the current expression is (PRINT Y), (MBD (COND ((NULL X) *) ((NULL (CAR Y)) * (GO LP))) would replace (PRINT Y) with (COND ((NULL X) (PRINT Y)) ((NULL (CAR Y)) (PRINT Y) (GO LP))).

(MBD e_1 ... e_m)	equivalent to (MBD (e_1 ... e_m *)).

Example: If the current expression is (PRINT Y), then (MBD SETQ X) will replace it with (SETQ X (PRINT Y)).

(MBD x)	x atomic, same as (MBD (x *)).

Example: If the current expression is (PRINT Y), (MBD RETURN) will replace it with (RETURN (PRINT Y)).

All three forms of MBD leave the edit chain so that the larger expression is the new current expression.

If the current expression *initially* is a tail, embedding works exactly the same as though the current expression were the first element in that tail. Thus if the current expression were ... (PRINT Y)(PRINT Z)), (MBD SETQ X) would replace (PRINT Y) with (SETQ X (PRINT Y)).

[†]a la subst, i.e., a fresh copy is used for each substitution.

The embed command can also incorporate a location specification.

(EMBED @ IN . x)[†] does (LC . @) and then (MBD . x). Edit chain is not changed, but unfind is set to the edit chain after the MBD was performed.

Example: (EMBED PRINT IN SETQ X), (EMBED 3 2 IN RETURN), (EMBED COND 3 1 IN (OR * (NULL X))).

WITH can be used for IN, and SURROUND can be used for EMBED, e.g., (SURROUND NUMBERP WITH (AND * (MINUSP X))).

[†] @ is the segment between EMBED and IN.

The MOVE Command

The MOVE command allows the user to specify (1) the expression to be moved, (2) the place it is to be moved to, and (3) the operation to be performed there, e.g., insert it before, insert it after, replace, etc.

(MOVE $@_1$ TO com . $@_2$)[†] where com is BEFORE, AFTER, or the name of a list command, e.g., :, N, etc. performs (LC . $@_1$), obtains the current expression there (or its first element, if it is a tail), let us call this expr; MOVE then goes back to original edit chain, performs (LC . $@_2$), performs (com expr),* then goes back to @1 and deletes expr. Edit chain is not changed. Unfind is set to edit chain after (com expr) was performed.

For example, if the current expression is (A B C D), (MOVE 2 TO AFTER 4) will make the new current expression be (A C D B). Note that 4 was executed as of the original edit chain, and that the second element had not yet been removed.

[†] @1 is the segment between MOVE and TO.

*Setting an internal flag so expr is not copied.

As the following examples taken from actual editing will show, the MOVE command is an extremely versatile and powerful feature of the editor.

```
*?
(PROG ((L L)) (EDLOC (CDDR C)) (RETURN (CAR L)))
*(MOVE 3 TO : CAR)
*?
(PROG ((L L)) (RETURN (EDLOC (CDDR C))))
*
```

```
*P
... (SELECTQ OBJPR & &) (RETURN &) LP2 (COND & &))
*(MOVE 2 TO N 1)
*P
... (SELECTQ OBJPR & & &) LP2 (COND & &))
```

```
*P
(OR (EQ X LASTAIL) (NOT &) (AND & & &))
*(MOVE 4 TO AFTER (BELOW COND))
*P
(OR (EQ X LASTAIL) (NOT &))
*\ P
... (& &) (AND & & &) (T & &))
*
```

```
*P
((NULL X) **COMMENT** (COND & &))
*(-3 (GO DELETE]
*(MOVE 4 TO N (← PROG))
*P
((NULL X) **COMMENT** (GO DELETE))
*\ P
(PROG (&) **COMMENT** (COND & & &) (COND & & &) (COND & &))
*(INSERT DELETE BEFORE -1)
*P
(PROG (&) **COMMENT** (COND & & &) (COND & & &) DELETE (COND & &))
*
```

Note that in the last example, the user could have added the prog label DELETE and moved the cond in one operation by performing (MOVE 4 TO N (← PROG) (N DELETE)). Similarly, in the next example, in the course of specifying @2, the location where the expression was to be moved to, the user also performs a structure modification, via (N (T)), thus creating the structure that will receive the expression being moved.

```
*P
((CDR &) **COMMENT** (SETQ CL &) (EDITSMASH CL & &))
*(MOVE 4 TO N Ø (N (T)) -1]
*P
((CDR &) **COMMENT** (SETQ CL &))
*\ P
*(T (EDITSMASH CL & &))
*
```

If @2 is NIL, or (HERE), the current position specifies where the operation is to take place. In this case, unfind is set to where the expression that was moved was originally located, i.e. $@_1$. For example:

```
*P
(TENEX)
*(MOVE ↑ F APPLY TO N HERE)
*P
(TENEX (APPLY & &))
*
```

```
*P
(PROG (& & & ATM IND VAL) (OR & &) **COMMENT** (OR & &) (PRIN1 & T) (
PRIN1 & T) (SETQ IND†

*(MOVE * TO BEFORE HERE)
*P
(PROG (& & & ATM IND VAL) (OR & &) (OR & &) (PRIN1 &†

*
```

†Sudden termination of output followed by an extra carriage return indicates printing was aborted by control-E. The * in (MOVE * TO BEFORE HERE) locates the comment, which is printed as **COMMENT**, see p. 9.68.

```
*P
(T (PRIN1 C-EXP T))
*(MOVE ↑ BF PRIN1 TO N HERE)
*P
(T (PRIN1 C-EXP T) (PRIN1 & T))
*
```

Finally, if @1 is NIL, the MOVE command allows the user to specify some place the *current expression* is to be moved to. In this case, the edit chain is changed, and is the chain where the current expression was moved to; unfind is set to where it was.

```
*P
(SELECTQ OBJPR (&) (PROGN & &))
*(MOVE TO BEFORE LOOP)
*P
... (SELECTQ OBJPR & &) LOOP (FRPLACA DFPRP &) (FRPLACD DFPRP
&) (SELECTQ
*
```

Commands That "Move Parentheses"

The commands presented in this section permit modification of the list structure itself, as opposed to modifying components thereof. Their effect can be described as inserting or removing a single left or right parenthesis, or pair of left and right parentheses. Of course, there will always be the same number of left parentheses as right parentheses in any list structure, since the parentheses are just a notational guide to the structure provided by print. Thus, no command can insert or remove just one parenthesis, but this is suggestive of what actually happens.

In all six commands, n and m are used to specify an element of a list, usually of the current expression. In practice, n and m are usually positive or negative integers with the obvious interpretation. However, all six commands use the generalized NTH command, p. 9.37, to find their element(s), so that nth element means the first element of the tail found by performing (NTH n). In other words, if the current expression is (LIST (CAR X) (SETQ Y (CONS W Z))), then (BI 2 CONS), (BI X -1), and (BI X Z) all specify the exact same operation.

All six commands generate an error if the element is not found, i.e. the NTH fails. All are undoable.

(BI n m)	both in, inserts a left parentheses before the nth element and after the mth element in the current expression. Generates an error if the mth element is not contained in the nth tail, i.e., the mth element must be "to the right" of the nth element.

Example: If the current expression is (A B (C D E) F G), then (BI 2 4) will modify it to be (A (B (C D E) F) G).

(BI n) same as (BI n n).

Example: If the current expression is (A B (C D E) F G), then (BI -2) will modify it to be (A B (C D E) (F) G).

(BO n) both out. Removes both parentheses from the nth element. Generates an error if nth element is not a list.

Example: If the current expression is (A B (C D E) F G), then (BO D) will modify it to be (A B C D E F G).

(LI n) left in, inserts a left parenthesis before the nth element (and a matching right parenthesis at the end of the current expression), i.e. equivalent to (BI n -1).

Example: If the current expression is (A B (C D E) F G), then (LI 2) will modify it to be (A (B (C D E) F G)).

(LO n) left out, removes a left parenthesis from the nth element. *All elements following the nth element are deleted.* Generates an error if nth element is not a list.

Example: If the current expression is (A B (C D E) F G), then (LO 3) will modify it to be (A B C D E).

(RI n m)	right in, inserts a right parenthesis after the mth element of the nth element. The rest of the nth element is brought up to the level of the current expression.

Example: If the current expression is (A (B C D E) F G), (RI 2 2) will modify it to be (A (B C) D E F G). Another way of thinking about RI is to read it as "move the right parenthesis at the end of the nth element *in* to after the mth element."

(RO n)	right out, removes the right parenthesis from the nth element, moving it to the end of the current expression. All elements following the nth element are moved inside of the nth element. Generates an error if nth element is not a list.

Example: If the current expression is (A B (C D E) F G), (RO 3) will modify it to be (A B (C D E F G)). Another way of thinking about RO is to read it as "move the right parenthesis at the end of the nth element *out* to the end of the current expression."

TO and THRU

EXTRACT, EMBED, DELETE, REPLACE, and MOVE can be made to operate on several contiguous elements, i.e., a segment of a list, by using the TO or THRU command in their respective location specifications.

(@1 THRU @2)	does a (LC . @1), followed by an UP, and then a (BI 1 @2), thereby grouping the segment into a single element, and finally does a 1, making the final current expression be that element.

For example, if the current expression is (A (B (C D) (E) (F G H) I) J K), following (C THRU G), the current expression will be ((C D) (E) (F G H)).

(@1 TO @2)	Same as THRU except last element not included, i.e., after the BI, an (RI 1 -2) is performed.

If both @1 and @2 are numbers, and @2 is greater than @1, then @2 counts from the beginning of the current expression, the same as @1. In other words, if the current expression is (A B C D E F G), (3 THRU 4) means (C THRU D), not (C THRU F). In this case, the corresponding BI command is (BI 1 @2-@1+1).

THRU to TO are not very useful commands by themselves, and are not intended to be used "solo", but in conjunction with EXTRACT, EMBED, DELETE, REPLACE, and MOVE. After THRU and TO have operated, they set an internal editor flag informing the above commands that the element they are operating on is actually a segment, and that the extra pair of parentheses should be removed when the operation is complete. Thus:

```
*P
(PROG (& & ATM IND VAL WORD) (PRIN1 & T) (PRIN1 & T) (SETQ IND &) (SETQ
VAL &) **COMMENT** (SETQQ

*(MOVE (3 THRU 4) TO BEFORE 7)
*P
(PROG (& & ATM IND VAL WORD) (SETQ IND &) (SETQ VAL &) (PRIN1 & T) (
PRIN1 & T) **COMMENT**

*


*P
(* FAIL RETURN FROM EDITOR. USER SHOULD NOTE THE VALUES OF SOURCEXPR AND
CURRENTFORM. CURRENTFORM IS THE LAST FORM IN SOURCEXPR WHICH WILL HAVE
BEEN TRANSLATED, AND IT CAUSED THE ERROR.)
*(DELETE (USER THRU CURR$))
=CURRENTFORM.
*P
(* FAIL RETURN FROM EDITOR. CURRENTFORM IS

*



*P
... LP (SELECTQ & & & & NIL) (SETQ Y &) OUT (SETQ FLG &) (RETURN Y))
*(MOVE (1 TO OUT) TO N HERE]
*P
... OUT (SETQ FLG &) (RETURN Y) LP (SELECTQ & & & & NIL) (SETQ Y &))
*
```

```
*PP
  [PROG (RF TEMP1 TEMP2)
        (COND
          ((NOT (MEMB REMARG LISTING))
            (SETQ TEMP1 (ASSOC REMARG NAMEDREMARKS))  **COMMENT**
            (SETQ TEMP2 (CADR TEMP1))
            (GO SKIP))
          (T   **COMMENT**
             (SETQ TEMP1 REMARG)))
        (NCONC1 LISTING REMARG)
        (COND
           ((NOT (SETQ TEMP2 (SASSOC


*(EXTRACT (SETQ THRU CADR) FROM COND)
*P
(PROG (RF TEMP1 TEMP2) (SETQ TEMP1 &) **COMMENT** (SETQ TEMP2 &)
(NCONC1 LISTING REMARG) (COND & &

*
```

TO and THRU can also be used directly with XTR.[†] Thus in the previous example, if the current expression had been the COND, e.g. the user had first performed F COND, he could have used (XTR (SETQ THRU CADR)) to perform the extraction.

[†]Because XTR involves a location specification while A,B,:, and MBD do not.

(@1 TO), (@1 THRU)	both same as (@1 THRU -1), i.e., from @1 thru the end of the list.

```
*P
(VALUE (RPLACA DFPRP &) (RPLACD &) (RPLACA VARSWORD &) (RETURN))
*(MOVE (2 TO) TO N (← PROG))
*(N (GO VAR))
*P
(VALUE (GO VAR))


*P
(T **COMMENT** (COND &) **COMMENT** (EDITSMASH CL & &) (COND &))
*(-3 (GO REPLACE))
*(MOVE (COND TO) TO N ↑ PROG (N REPLACE))
*P
(T **COMMENT** (GO REPLACE))
*\ P
(PROG (&) **COMMENT** (COND & & &) (COND & & &) DELETE (COND & &)
REPLACE (COND &) **COMMENT** (EDITSMASH CL & &) (COND &))
*


*PP
  [LAMBDA (CLAUSALA X)
    (PROG (A D)
          (SETQ A CLAUSALA)
      LP  (COND
            ((NULL A)
              (RETURN)))
          (SERCH X A)
          (RUMARK (CDR A))
          (NOTICECL (CAR A))
          (SETQ A (CDR A))
          (GO LP]
*(EXTRACT (SERCH THRU NOT$) FROM PROG)
=NOTICECL
*P
(LAMBDA (CLAUSALA X) (SERCH X A) (RUMARK &) (NOTICECL &))
*(EMBED (SERCH TO) IN (MAP CLAUSALA (FUNCTION (LAMBDA (A) *]
*PP
  [LAMBDA (CLAUSALA X)
    (MAP CLAUSALA (FUNCTION (LAMBDA (A)
                 (SERCH X A)
                 (RUMARK (CDR A))
                 (NOTICECL (CAR A]
*
```

(R x y) replaces all instances of x by y in the current expression, e.g., (R CAADR CADAR). Generates an error if there is not a least one instance.

R operates by performing a dsubst. The current expression is the third argument to dsubst, i.e., the expression being substituted into, and y is the first argument to dsubst, i.e., the expression being substituted. R computes the second argument to dsubst, the expression to be substituted *for*, by performing (F x T). The second argument is then the current expression at that point, or if that current expression is a list and x is atomic, then the first element of that current expression. Thus x can be the S-expression (or atom) to be substituted for, *or* can be a pattern which specifies that S-expression (or atom).

For example, if the current expression is
(LIST FUNNYATOM1 FUNNYATOM2 (CAR FUNNYATOM1)), then
(R FUN$ FUNNYATOM3) will substitute FUNNYATOM3 for FUNNYATOM1 throughout the current expression. Note that FUNNYATOM2, even though it would have matched with the pattern FUN$, is *not* replaced.

Similarly, if (LIST (CAR X)(CAR Y)) is the first expression matched by (LIST --), then (R (LIST --) (LIST (CAR Y) (CAR Z))) is equivalent to (R (LIST (CAR X) (CAR Y)) (LIST (CAR Y) (CAR Z))), i.e., both will replace all instances of (LIST (CAR X) (CAR Y)) by (LIST (CAR Y) (CAR Z)). Note that other forms beginning with LIST will *not* be replaced, even though they would have matched with (LIST --). To change *all* expressions of the form (LIST --) to (LIST (CAR Y) (CAR Z)), the user should perform (LP (REPLACE (LIST --) WITH (LIST (CAR Y) (CAR].

unfind is set to the edit chain following the find command so that \ will make the current expression be the place where the first substitution occurred.

(SW n m) switches the _nth_ and _mth_ elements of the current expression.

For example, if the current expression is
(LIST (CONS (CAR X) (CAR Y)) (CONS (CDR X) (CDR Y))),
(SW 2 3) will modify it to be
(LIST (CONS (CDR X) (CDR Y)) (CONS (CAR X) (CAR Y))). The relative order of _n_ and _m_ is not important, i.e., (SW 3 2) and (SW 2 3) are equivalent.

SW uses the generalized NTH command to find the _nth_ and _mth_ elements, a la the BI-BO commands.

Thus in the previous example, (SW CAR CDR) would produce the same result.

Commands That Print

P	prints current expression as though printlevel were set to 2.
(P m)	prints mth element of current expression as though printlevel were set to 2.
(P 0)	same as P
(P m n)	prints mth element of current expression as though printlevel were set to n.
(P 0 n)	prints current expression as though printlevel were set to n.
?	same as (P Ø 100)

Both (P m) and (P m n) use the general NTH command to obtain the corresponding element, so that m does not have to be a number, e.g. (P COND 3) will work.

All printing functions print to the teletype, regardless of the primary output file. No printing function ever changes the edit chain. All record the current edit chain for use by \P, p. 9.40 All can be aborted with Control-E. PP causes all comments to be printed as **COMMENT** (see p. 14.26). P and ? print as **COMMENT** only those comments that are (top level) elements of the current expression.†

†Lower expressions are not seen; the printing command simply sets printlevel and calls print.

Commands That Evaluate

E	*only when typed in*,[*] causes the editor to call lispx giving it the next input as argument.[**]

```
Example: *E BREAK(FIE FUM)
         (FIE FUM)
         *E (FOO)

              (FIE BROKEN)

         :
```

(E x)	evaluates x, i.e., performs eval[x], and prints the result on the teletype.
(E x T)	same as (E x) but does not print.

The (E x) and (E x T) commands are mainly intended for use by macros and subroutine calls to the editor; the user would probably type in a form for evaluation using the more convenient format of the (atomic) E command.

*i.e. (INSERT D BEFORE E) will treat E as a pattern.

** lispx is used by evalqt and break for processing teletype inputs. If nothing else is typed on the same line, lispx evaluates its argument. Otherwise, lispx applies it to the next input. In both cases, lispx prints the result. See example, and Sections 2 and 22.

(I c x_1 ... x_n)   same as (c y_1 ... y_n) where y_i=eval[x_i].

Example: (I 3 (GETD (QUOTE FOO)) will replace the 3rd element of the current expression with the definition of foo.*
(I N FOO (CAR FIE)) will attach the value of foo and car of the value of fie to the end of the current expression.
(I F= FOO T) will search for an expression eq to the value of foo.

If c is not an atom, it is evaluated as well.

Example: (I (COND ((NULL FLG) (QUOTE -1)) (T 1)) FOO), if flg is NIL, inserts the value of foo before the first element of the current expression, otherwise replaces the first element by the value of foo.

##[com_1;com_2; ... ;com_n] is an NLAMBDA, NOSPREAD function (not a command). Its value is what the current expression would be after executing the edit commands com_1 ... com_n starting from the present edit chain. Generates an error if any of com_1 thru com_n cause errors. The current edit chain is never changed.**

Example: (I R (QUOTE X) (## (CONS .. Z))) replaces all X's in the current expression by the first cons containing a Z.

---

*The I command sets an internal flag to indicate to the structure modification commands *not* to copy expression(s) when inserting, replacing, or attaching.

**Recall that A,B,:,INSERT, REPLACE, and CHANGE make special checks for ## forms in the expressions used for inserting or replacing, and use a copy of ## form instead (see p. 9.48). Thus, (INSERT (## 3 2) AFTER 1) is equivalent to (I INSERT (COPY (## 3 2)) (QUOTE AFTER) 1).

The I command is not very convenient for computing an *entire* edit command for execution, since it computes the command name and its arguments separately. Also, the I command cannot be used to compute an atomic command. The following two commands provide more general ways of computing commands.

(COMS $x_1, \ldots, x_n$)	Each x_i is evaluated and its value executed as a command.

For example, (COMS (COND (X (LIST 1 X)))) will replace the first element of the current expression with the value of $\underline{x}$ if non-NIL, otherwise do nothing.*

(COMSQ $com_1, \ldots, com_n$) executes $com_1, \ldots, com_n$.

COMSQ is mainly useful in conjunction with the COMS command. For example, suppose the user wishes to compute an entire list of commands for evaluation, as opposed to computing each command one at a time as does the COMS command. He would then write (COMS (CONS (QUOTE COMSQ) x)) where x computed the list of commands, e.g.,
(COMS (CONS (QUOTE COMSQ) (GETP FOO (QUOTE COMMANDS)))).

*NIL as a command is a NOP, see p. 9.78.

Commands That Test

(IF x) generates an error *unless* the value of eval[x] is true, i.e., if eval[x] causes an error or eval[x]=NIL, IF will cause an error.

For some editor commands, the occurrence of an error has a well defined meaning, i.e., they use errors to branch on as cond uses NIL and non-NIL. For example, an error condition in a location specification may simply mean "not this one, try the next." Thus the location specification (IPLUS (E (OR (NUMBERP (## 3)) (ERROR!)) T)) specifies the first IPLUS whose second argument is a number. The IF command, by equating NIL to error, provides a more natural way of accomplishing the same result. Thus, an equivalent location specification is (IPLUS (IF (NUMBERP (## 3)))).

The IF command can also be used to select between two alternate lists of commands for execution.

(IF x coms_1 coms_2) If eval[x] is true, execute coms_1; if eval[x] causes an error or is equal to NIL, execute coms_2.†

For example, the command (IF (READP T) NIL (P)) will print the current expression provided the input buffer is empty.

IF can also be written as

(IF x coms_1) if eval[x] is true, execute coms_1; otherwise generate an error.

†Thus IF is equivalent to (COMS (CONS (QUOTE COMSQ) (COND ((CAR (NLSETQ (EVAL X))) COMS1) (T COMS2)))).

(LP . coms) repeatedly executes coms, a list of commands, until an error occurs.

For example, (LP F PRINT (N T)) will attach a T at the end of every print expression. **(LP F PRINT (IF (## 3) NIL ((N T))))** will attach a T at the end of each print expression which doe not already have a second argument.*

When an error occurs, LP prints n OCCURRENCES. where n is the number of times coms was successfully executed. The edit chain is left as of the last complete successful execution of coms.

(LPQ . coms) Same as LP but does not print n OCCURRENCES.

In order to prevent non-terminating loops, both LP and LPQ terminate when the number of iterations reaches maxloop, initially set to 30. Since the edit chain is left as of the last successful completion of the loop, the user can simply continue the LP command with REDO (Section 22).

---

*i.e. the form (## 3) will cause an error if the edit command 3 causes an error, thereby selecting ((N T)) as the list of commands to be executed. The IF could also be written as (IF (CDDR (##)) NIL ((N T))).

(ORR $coms_1$,...,$coms_n$) ORR begins by executing $coms_1$, a list of commands. If no error occurs, ORR is finished. Otherwise, ORR restores the edit chain to its original value, and continues by executing $coms_2$, etc. If none of the command lists execute without errors, i.e., the ORR "drops off the end", ORR generates an error. Otherwise, the edit chain is left as of the completion of the first command list which executes without an error.*

For example, (ORR (NX) (!NX) NIL) will perform a NX, if possible, otherwise a !NX, if possible, otherwise do nothing. Similarly, DELETE could be written as (ORR (UP (1)) (BK UP (2)) (UP (: NIL))).

* NIL as a command list is perfectly legal, and will always execute successfully. Thus, making the last 'argument' to ORR be NIL will insure that the ORR never causes an error. Any other atom is treated as (atom), i.e., the above example could be written as (ORR NX !NX NIL).

Macros

Many of the more sophisticated branching commands in the editor, such as ORR, IF, etc., are most often used in conjunction with edit macros. The macro feature permits the user to define new commands and thereby expand the editor's repertoire.* Macros are defined by using the M command.

(M c . coms)	For c an atom, M defines c as an atomic command.** Executing c is then the same as executing the list of commands coms.

For example, (M BP BK UP P) will define BP as an atomic command which does three things, a BK, an UP, and a P. Note that macros can use commands defined by macros as well as built in commands in their definitions. For example, suppose Z is defined by (M Z -1 (IF (READP T) NIL (P))), i.e. Z does a -1, and then if nothing has been typed, a P. Now we can define ZZ by (M ZZ -1 Z), and ZZZ by (M ZZZ -1 -1 Z) or (M ZZZ -1 ZZ).

Macros can also define list commands, i.e., commands that take arguments.

(M (c) (arg_1, ..., arg_n) . coms)	c an atom. M defines c as a list command. Executing (c e_1, ..., e_n) is then performed by substituting e_1 for arg_1, ..., e_n for arg_n throughout coms, and then executing coms.

For example, we could define a more general BP by (M (BP) (N) (BK N) UP P). Thus, (BP 3) would perform (BK 3), followed by an UP, followed by a P.

---

* However built in commands always take precedence over macros, i.e., the editor's repertoire can be expanded, but not modified.

** If a macro is redefined, it new definition replaces its old.

A list command can be defined via a macro so as to take a fixed or indefinite number of 'arguments', i.e., be spread or nospread. The form given above specified a macro with a fixed number of arguments, as indicated by its argument list. If the 'argument list' is *atomic*, the command takes an indefinite number of arguments.*

(M (c) args . coms)	name, args both atoms, defines c as a list command. Executing (c $e_1, \ldots, e_n$) is performed by substituting ($e_1, \ldots, e_n$), i.e., cdr of the command, for args throughout coms, and then executing coms.

For example, the command 2ND, p. 9.35 , can be defined as a macro by (M (2ND) X (ORR ((LC . X) (LC . X)))).

Note that for all editor commands, 'built in' commands as well as commands defined by macros, atomic definitions and list definitions are *completely independent*. In other words, the existence of an atomic definition for c in *no* way affects the treatment of c when it appears as car of a list command, and the existence of a list definition for c in *no* way affects the treatment of c when it appears as an atom. In particular, c can be used as the name of either an atomic command, or a list command, or both. In the latter case, two entirely different definitions can be used.

Note also that once c is defined as an atomic command via a macro definition, it will *not* be searched for when used in a location specification, unless it is preceded by an F. Thus (INSERT -- BEFORE BP) would not search for BP, but instead perform a BK, an UP, and a P, and then do the insertion. The corresponding also holds true for list commands.

* Note parallelism to EXPR's and EXPR*'s.

Occasionally, the user will want to employ the S command in a macro to save some temporary result. For example, the SW command could be defined as

```
(M (SW) (N M) (NTH N) (S FOO 1) MARK Ø (NTH M) (S FIE 1)
   (I 1 FOO) ←← (I 1 FIE))†
```

Since SW sets foo and fie, using SW may have undesirable side effects, especially when the editor was called from deep in a computation. Thus we must always be careful to make up unique names for dummy variables used in edit macros, which is bothersome. Furthermore, it would be impossible to define a command that called itself recursively while setting free variables. The BIND command solves both problems.

(BIND . coms)	binds three dummy variables #1, #2, #3, (initialized to NIL), and then executes the edit commands coms. Note that these bindings are only in effect while the commands are being executed, and that BIND can be used recursively; it will rebind #1, #2, and #3 each time it is invoked.††

Thus we could now write SW safely as

```
(M (SW) (N M) (BIND (NTH N) (S #1 1) MARK Ø (NTH M) (S #2 1)
        (I 1 #1) ←← (I 1 #2))).
```

User macros are stored on a list usermacros.††† Thus if the user wants to save his macros, he should save the value of usermacros.††††

† A more elegant definition would be
(M (SW) (N M) (NTH N) MARK Ø (NTH M) (S FIE 1) (I 1 (## ← 1))
   ←← (I 1 FIE)), but this would still use one free variable.

†† BIND is implemented by (PROG (#1 #2 #3) (EDITCOMS (CDR COM))) where com corresponds to the BIND command, and editcoms is an internal editor function which executes a list of commands.

††† usermacros is initially NIL.

††††The user probably should also save the value of editcomsa and editcomsl, see p. 9.85.

Miscellaneous Commands

NIL	unless preceded by F or BF, is always a NOP. Thus extra right parentheses or square brackets at the ends of commands are ignored.
TTY:	calls the editor recursively. The user can then type in commands, and have them executed. The TTY: command is completed when the user exits from the lower editor. (See OK and STOP below).

The TTY: command is _extremely_ useful. It enables the user to set up a complex operation, and perform interactive attention-changing commands part way through it. For example the command (MOVE 3 TO AFTER COND 3 P TTY:) allows the user to interact, in effect, *within* the MOVE command. Thus he can verify for himself that the correct location has been found, or complete the specification "by hand." In effect, TTY: says "I'll tell you what you should do when you get there."

The TTY: command operates by printing TTY: and then calling the editor. The initial edit chain in the lower editor is the one that existed in the higher editor at the time the TTY: command was entered. Until the user exits from the lower editor, any attention changing commands he executes only affect the lower editor's edit chain.* When the TTY: command finishes, the lower editor's edit chain becomes the edit chain of the higher editor.

*Of course, if the user performs any structure modification commands while under a TTY: command, these will modify the structure in both editors, since it is the same structure.

OK	exits from the editor
STOP	exits from the editor with an error. Mainly for use in conjunction with TTY: commands that the user wants to abort.

Since all of the commands in the editor are errorset protected, the user must exit from the editor via a command.[†] STOP provides a way of distinguishing between a successful and unsuccessful (from the user's standpoint) editing session. For example, if the user is executing (MOVE 3 TO AFTER COND TTY:), and he exits from the lower editor with an OK, the MOVE command will then complete its operation. If the user wants to abort the MOVE command, he must make the TTY: command generate an error. He does this by exiting from the lower editor with a STOP command. In this case, the higher editor's edit chain will not be changed by the TTY: command.

SAVE	exits from the editor and saves the 'state of the edit' on the property list of the function/variable being edited under the property EDIT-SAVE. If the editor is called again on the same structure, the editing is effectively "continued," i.e., the edit chain, mark list, value of <u>unfind</u> and <u>undolst</u> are restored.

For example:

```
*P
(NULL X)
* F COND P
(COND (& &) (T &))
SAVE
FOO
 .
 .
 .
←EDIT(FOO)
EDIT
*P
(COND (& &) (T &))
*\ P
(NULL X)
*
```

[†]Or by typing a control-D. STOP is preferred even if the user is editing at the <u>evalqt</u> level, as it will perform the necessary 'wrapup' to insure that the changes made while editing will be undoable (see Section 22).

SAVE is necessary only if the user is editing many different expressions, an exit from the editor via OK always saves the state of the edit of that call to the editor.[†] Whenever the editor is entered, it checks to see if it is editing the same expression as the last one edited. In this case, it restores the mark list, the undolst, and sets <u>unfind</u> to be the edit chain as of the previous exit from the editor. For example:

```
←EDITF(FOO)
EDIT
*P
(LAMBDA (X) (PROG & & LP & & & &))
    .
    .
    .
*P
(COND & &)
*OK
FOO
←
    .
    .                        any number of evalqt inputs except for
    .                        calls to the editor
←EDITF(FOO)
EDIT
*P
(LAMBDA (X) (PROG & & LP & & & &))
*\ P
(COND & &)
*
```

Furthermore, as a result of the history feature (Section 22), if the editor is called on the same expression within a certain number of <u>evalqt</u> inputs,[††] the state of the edit of that expression is restored, regardless of how many other expressions may have been edited in the meantime.

[†]On the property list of the atom EDIT, under the property name LASTVALUE. OK also remprops EDIT-SAVE from the property list of the function/variable being edited.

[††] Namely, the size of the history list, initially 30, but it can be increased by the user.

For example:

```
←EDITF(FOO)
EDIT
*
   .
   .
   .
*P
(COND (& &) (& &) (&) (T &))
*OK
FOO
←
   .              less than 30 evalqt inputs, including editing
   .
   .
←EDITF(FOO)
EDIT
*\ P
(COND (& &) (& &) (&) (T &))
*
```

Thus the user can always continue editing, including undoing changes from a previous editing session, if

(1) No other expressions have been edited since that session;[†] or

(2) That session was 'sufficiently' recent; or

(3) It was ended with a SAVE command.

[†] Since saving takes place at *exit* time, intervening calls that were aborted via control-D or exited via STOP will not affect the editor's memory of this last session.

%%,RAISE,LOWER,CAP	Used for editing lower case comments. See pp. 14.39-14.40.
REPACK	Permits the 'editing' of an atom or string.

For example:

```
*P
... "THIS IS A LOGN STRING")
REPACK
*EDIT
P
(T H I S %  I S %  A %  L O G N %  S T R I N G)
*(SW G N)
*OK
"THIS IS A LONG STRING"
*
```

REPACK operates by calling the editor recursively on unpack of the current expression, or if it is a list, on unpack of its first element. If the lower editor is exited successfully, i.e. via OK as opposed to STOP, the list of atoms is made into a single atom or string, which replaces the atom or string being 'repacked.' The new atom or string is always printed.

(REPACK @)	does (LC . @) followed by REPACK, e.g. (REPACK THIS$).

UNDO

Each command that causes structure modification automatically adds an entry to the front of undolst containing the information required to restore all pointers that were changed by the command.

UNDO	undoes the last, i.e. most recent, structure modification command that has not yet been undone,* and prints the name of that command, e.g., MBD UNDONE. The edit chain is then *exactly* what it was before the 'undone' command had been performed. If there are no commands to undo, UNDO types NOTHING SAVED.
!UNDO	undoes all modifications performed during this editing session, i.e. this call to the editor. As each command is undone, its name is printed a la UNDO. If there is nothing to be undone, !UNDO prints NOTHING SAVED.

Whenever the user *continues* an editing session as described on pages 9.79-9.81, the undo information of the previous session(s) is protected by inserting a special blip, called an undo-block on the front of undolst. This undo-block will terminate the operation of a !UNDO, thereby confining its effect to the current session, and will similarly prevent an UNDO command from operating on commands executed in the previous session.

*Since UNDO and !UNDO causes structure modification, they also add an entry to undolst. However, UNDO and !UNDO entries are skipped by UNDO, e.g., if the user performs an INSERT, and then an MBD, the first UNDO will undo the MBD, and the second will undo the INSERT. However, the user can also specify precisely which commands he wants undone by identifying the corresponding entry on the history list as described in Chapter 22. In this case, he can undo an UNDO command, e.g. by typing UNDO UNDO, or undo a !UNDO command, or undo a command other than that most recently performed.

Thus, if the user enters the editor continuing a session, and immediately executes an UNDO or !UNDO, UNDO and !UNDO will type BLOCKED, instead of NOTHING SAVED. Similarly, if the user executes several commands and then undoes them all, either via several UNDO commands or a !UNDO command, another UNDO or !UNDO will also type BLOCKED.

UNBLOCK	removes an undo-block. If executed at a non-blocked state, i.e. if UNDO or !UNDO *could* operate, types NOT BLOCKED.
TEST	adds an undo-block at the front of <u>undolst</u>.

Note that TEST together with !UNDO provide a 'tentative' mode for editing, i.e. the user can perform a number of changes, and then undo all of them with a single !UNDO command.

Editdefault

Whenever a command is not recognized, i.e., is not 'built in' or defined as a macro, the editor calls an internal function, editdefault to determine what action to take. If a location specification is being executed, an internal flag informs editdefault to treat the command as though it had been preceded by an F.

If the command is a list command, an attempt is made to perform spelling correction on car of the command* using editcomsl, a list of all list edit commands.** If spelling correction is successful,*** the correct command name is rplacaed into the command, and the editor continues by executing the command. In other words, if the user types (LP F PRINT (MBBD AND (NULL FLG))), only one spelling corrections will be necessary to change MBBD to MBD. If spelling correction is not successful, an error is generated.

If the command given to editdefault is atomic, but was *not* typed in directly, the procedure is similar to that of list commands, namely to attempt spelling correction using editcomsa, a list of atomic edit commands, and if successful, make the change in the command list that the atomic command came from, and then continue. Thus (LP FF PRINT (MBD AND (NULL FLG))) will require only one spelling correction to change FF to F.

---

* unless dwimflg=NIL. See section 17 for discussion of spelling correction.

** When a macro is defined via the M command, the command name is added to editcomsa or editcomsl, depending on whether it is an atomic or list command. Thus if the user saves his macro definitions by dumping the value of usermacros, he should also save editcomsl and editcomsa if he wants spelling correction to include his macros, or if he wants to use the 'line command format', described on next page, with macros.

*** If the command was not typed in directly, the user will be asked to approve the correction. See section 17.

If the command is atomic *and* typed in directly, the procedure followed is a little more elaborate.

1) If the command is one of the list commands, i.e., a member of editcomsl, and there is additional input on the same teletype line, treat the entire line as a single list command.† Thus, the user may omit parentheses for any list command typed in at the top level (which is not also an atomic command, e.g. NX, BK). For example

```
*P
(COND (& &) (T &))
*XTR 3 2)
*MOVE TO AFTER LP
*
```

If the command is on the list editcomsl but no additional input is on the teletype line, an error is generated, e.g.

```
*P
(COND (& &) (T &))
*MOVE

MOVE ?
*
```

2) If the first character in the command is an 8, treat the 8 as a mistyped left parenthesis, and the rest of the line as the arguments to the command, e.g.,

```
*P
(COND (& &) (T &))
8-2 (Y (RETURN Z)))
*P
(COND (Y &) (& &) (T &))
```

† uses readline, p. 14.11. Thus the line can be terminated by carriage return, right parenthesis or square bracket, or a list.

3) If the last character in the command is P, and the first n-1 characters comprise the command ←, ↑, UP, NX, BK, !NX, UNDO, REDO, or a number, assume that the user intended two commands, e.g.,

```
*P
(COND (& &) (T &))
*ØP
=Ø P
(SETQ X (COND & &))
```

4) Otherwise, spelling correct using editcomsa, and if successful,† execute the corrected command.

5) Otherwise, if there is additional input on the same line, spelling correct using editcomsl, e.g.,

```
*MBBD SETQ X
=MBD
```

6) Otherwise, generate an error.

† No approval necessary since command was typed in directly.

Editor Functions

edite[expr;coms;atm] — edits an expression. Its value is the last element of editl[list[expr];coms;atm]. Generates an error if expr is not a list.

editl[l;coms;atm;marklst;mess] — editl[†] *is* the editor. Its first argument is the edit chain, and its value is an edit chain, namely the value of l at the time editl is exited.*

coms is an optional list of commands. For interactive editing, coms is NIL. In this case, editl types EDIT and then waits for input from the teletype.** Exit occurs only via an OK, STOP, or SAVE command.

If coms is *not* NIL, no message is typed, and each member of coms is treated as a command and executed. If an error occurs in the execution of one of the commands, no error message is printed, the rest of the commands are ignored, and editl exits with an error, i.e. the effect is the same as though a STOP command had been executed. If all commands execute successfully, editl returns the current value of l.

marklst is the list of marks.

† edit-*ell*, not edit-*one*.

*l is a specvar, and so can be examined or set by edit commands. For example, ↑ is equivalent to (E (SETQ L(LAST L)) T)

**If mess is not NIL, editl types it instead of EDIT. For example, the TTY: command is essentially (SETQ L (EDITL L NIL NIL NIL (QUOTE TTY:))).

atm is optional. On calls from *editf*, it is the name of the function being edited; on calls from *editv*, the name of the variable, and calls from *editp*, the atom whose property list is being edited. The property list of *atm* is used by the SAVE command for saving the state of the edit. Thus SAVE will not save anything if *atm*=NIL i.e. when editing arbitrary expressions via *edite* or *editl* directly. Furthermore, if *atm*=NIL, *editl* does not search the history list looking for a previous call to the editor, as described on p. 9.80.

editf[x]

nlambda, nospread function for editing a function. car[x] is the name of the function, cdr[x] an optional list of commands. For the rest of the discussion, fn is car[x], and coms is cdr[x].

The value of editf is fn.

(1) In the most common case, fn is an expr, and editf simply performs putd[fn;edite[getd[fn]; coms;fn]].

(2) If fn is not an expr, but has an EXPR property, editf prints PROP, and performs edite[getp[fn;EXPR];coms;fn]. When edite returns, if the editing is not terminated by a STOP, editf does unsavedef[fn], prints UNSAVED, and then does putd[fn; value-of-edite].

(3) If fn is neither an expr nor has an EXPR property, but its top level value is a list, editf assumes the user meant to call editv, prints =EDITV, calls editv and returns. Similarly, if fn has a non-NIL property list, editf prints =EDITP, calls editp and returns.

(4) If fn is neither a function, nor has an EXPR property, nor a top level value that is a list, nor a non-NIL property list, editf attempts spelling correction on the list userwords,* and if successful, goes back to (1).

Otherwise, editf generates an fn NOT EDITABLE error.

*Unless dwimflg=NIL. Spelling correction is via the function misspelled? If fn=NIL, misspelled returns the last 'word' referenced, e.g. by defineq, editf, prettyprint etc. Thus if the user defines foo and types editf[], the editor will assume he meant foo, type =FOO, and then type EDIT. See Section 17.

If editf ultimately succeeds in finding a function to edit, i.e. does not exit by calling editv or editp, **editf calls** the function addspell after editing has been completed.* Addspell 'notices' fn, i.e. sets lastword to fn, and adds fn to the appropriate spelling lists. editf also calls newfile?, which performs the updating for the file package as described on p. 14.41.

*Unless dwimflg=NIL. addspell is described in Section 17.

editv[editvx]   nlambda, nospread function, similar to editf, for editing values. car[editvx] specifies the value, cdr[editvx] is an optional list of commands.

If car[editvx] is a list, it is evaluated and its value given to edite, e.g. EDITV((CDR (ASSOC (QUOTE FOO) DICTIONARY))). In this case, the value of editv is T.

However, in most cases, car[editvx] is a variable, e.g. EDITV(FOO); and editv calls edite on the value of the variable.

If the value of car[editvx] is NOBIND, editv first attempts spelling correction using the list userwords.* Then editv will call edite on the value of car[editvx] (or the corrected spelling thereof). Thus, if the value of foo is NIL, and the user performs (EDITV FOO), no spelling correction will occur, since foo is the name of a variable in the user's system, i.e. it has a value. However, edite will generate an error, since foo's value is not a list, and hence not editable. If the user performs (EDITV FOOO), where the value of fooo is NOBIND, and foo is on the user's spelling list, the spelling corrector will correct FOOO to FOO. Then edite will be called on the value of foo. Note that this may still result in an error if the value of foo is not a list.

When (if) edite returns, editv sets the variable to the value returned, and calls addspell and newfile?.

The value of editv is the name of the variable whose value was edited.

*Unless dwimflg=NIL. Spelling correction is also performed if car[editvx] is NIL, so that EDITV() will edit lastword.

editp[x]

nlambda, nospread function, similar to editf for editing property lists. If the property list of car[x] is NIL, editp attempts spelling correction using userwords. Then editp calls edite on the propertv list of car[x], (or the corrected spelling thereof). When (if) edite returns, editp rplacd's car[x] with the value returned, and calls addspell.

The value of editp is the atom whose property list was edited.

editfns[x] — nlambda, nospread function, used to perform the same editing operations on several functions. car[x] is evaluated to obtain a list of functions. cdr[x] is a list of edit commands. editfns maps down the list of functions, prints the name of each function, and calls the editor (via editf) on that function.*

For example, EDITFNS(FOOFNS (R FIE FUM)) will change every FIE to FUM in each of the functions on foofns.

The call to the editor is errorset protected, so that if the editing of one function causes an error, editfns will proceed to the next function.**

Thus in the above example, if one of the functions did not contain a FIE, the R command would cause an error, but editing would continue with the next function.

The value of editfns is NIL

---

*i.e. the definition of editfns might be

```
(MAPC (EVAL (CAR X)) (FUNCTION (LAMBDA (Y)
      (APPLY (QUOTE EDITF)
             (CONS (PRINT Y T) (CDR X]
```

**In particular, if that function was being edited via its EXPR property, it will not be unsaved. In other words, only those functions for which the commands are successfully completed are unsaved. Thus, in our example, only those functions which contained a FIE, i.e. only those actually changed would be unsaved.

edit4e[pat;y] is the pattern match routine. Its value is T if pat matches y. See pp. 9.24-25 for definition of 'match'.

Note: before each search operation in the editor begins, the entire pattern is scanned for atoms or strings that end in alt-modes. These are replaced by patterns of the form (CONS (QUOTE $) (CHCON atom/string)). Thus from the standpoint of edit4e, pattern type 6, atoms or strings ending in alt-modes, is really "If car[pat] is the atom $ (alt-mode), pat will match with any literal atom or string whose initial character codes* are the same as those in cdr[pat]."

If the user wishes to call edit4e directly, he must therefore convert any patterns which contain atoms or strings ending in alt-modes to the form recognized by edit4e. This can be done via the function editfpat.

editfpat[pat;flg] makes a copy of pat with all patterns of type 6 converted to the form expected by edit4e.**

editfindp[x;pat;flg] allows a program to use the edit find command as a pure predicate from outside the editor. x is an expression, pat a pattern. The value of editfindp is T if the command F pat *would* succeed, NIL otherwise. editfindp calls editfpat to convert pat to the form expected by edit4e, unless flg=T. Thus, if the program is applying editfindp to several different expressions using the same pattern, it will be more efficient to call editfpat once, and then call editfindp with the converted pattern and flg=T.

*Up to the 27, the character code for $.

**flg=T is for internal use by the editor.

esubst[x;y;z;flg]

equivalent to performing (R y x) with z as the current expression, i.e. the order of arguments is the same as for subst. The value of esubst is the modified z. Generates an error* if y not found in z. If flg=T, also prints an error message of the form y ?. See p. 6.7.

changename[fn;from;to]

replaces all occurrences of from by to in the definition of fn. If fn is an expr, changename performs nlsetq[esubst[to;from;getd[fn]]]. If fn is compiled, changename searches the literals of fn (and all of its compiler generated subfunctions), replacing each occurrence of from with to.**

The value of changename is fn if at least one instance of from was found, otherwise NIL.

changename is used by break and advise for changing calls to fn1 to calls to fn1-IN-fn2.

---

*of the type that never causes a break.

**Will succeed even if from is called from fn via a linked call. In this case, the call will also be relinked to call to instead.

editracefn[com] is available to help the user debug complex edit macros, or subroutine calls to the editor. editracefn is to be defined by the user. Whenever the *value* of editracefn is T, the editor calls the *function* editracefn before executing each command (at any level), giving it that command as its argument.

For example, defineing editracefn as (LAMBDA (C) (PRINT C) (PRINT (CAR L))) will print each command and the corresponding current expression. (LAMBDA (C) (BREAK1 T T)) will cause a break before executing each command.

editracefn is initially equal to NIL, and undefined.

SECTION X

ATOM, STRING, ARRAY, AND STORAGE MANIPULATION

Contents

Atom Manipulation

The term 'print name' (of an atom) in LISP 1.5 referred to the characters that were output whenever the atom was printed. Since these characters were stored on the atom's property list under the property PNAME, pname was used interchangeably with 'print name'. In BBN-LISP, all pointers have pnames, although only literal atoms and strings have their pname explicitly stored.

The pname of a pointer is those characters that are output when the pointer is printed using prin1,

e.g. the pname of the atom ABC%(D[†] consists of the five characters ABC(D. The pname of the list (A B C) consists of the seven characters (A B C) (two of the characters are spaces).

Sometimes we will have occasion to refer to the prin2-pname.

The prin2-pname are those characters output when the corresponding pointer is printed using prin2.

Thus the prin2-pname of the atom ABC%(D is the *six* characters ABC%(D. Note that the pname of numbers depends on the setting of radix.

† % is the escape character. See sections 2 and 14.

pack[x]

If x is a list of atoms, the value of pack is a single atom whose pname is the concatenation of the pnames of the atoms in x, e.g.

```
   pack[(A BC DEF G)]=ABCDEFG
```

Although x is usually a list of atoms, it can be a list of arbitrary LISP pointers. The value of pack is still a single atom whose pname is the same as the concatenation of the pnames of all the pointers in x. e.g.

```
   pack[(1 "3.4" 5)] = 13.45,
      a floating point number
   pack[(A (B C) D)] = A%(B% C%)D,
```

In other words, mapc[x;prinl] and prinl[pack[x]] produce *exactly* the same output. In fact, pack actually operates by calling prinl to convert the pointers to a stream of characters (without printing) and then makes an atom out of the result.

Note however that atoms are restricted to < 99 characters. Attempting to create a larger atom either via pack or by typing one in (or reading from a file) will cause an error.

unpack[x;flg]

The value of unpack is the pname of x as a list of characters (atoms),[*] e.g.

```
    unpack[ABC] = (A B C)
    unpack["ABC(D"] = (A B C %( D)
```

In other words prin1[x] and mapc[unpack[x];prin1] produce the same output. If flg=T, the prin2-pname of x is used, e.g.

```
 unpack["ABC(D" ; T]=
            (%" A B C %( D %" )
```

Note that unpack performs n conses, where n is the number of characters in the pname of x.

dunpack[x;scratchlist;flg]

a destructive unpack that uses scratchlist to make a list equal to unpack[x;flg]. If the p-name is too long to fit in scratchlist, dunpack returns unpack[x;flg]. Gives error if scratchlist is not a list.

nchars[x]

number of characters in pname of x.[**]

---

*There are no special 'character-atoms' in BBN-LISP, i.e. an atom consisting of a single character is the same as any other atom.

**Both nthchar and nchars work much faster on objects that actually have an internal representation of their pname, i.e. literal atoms and strings, as they do not have to simulate printing.

nthchar[x;n]	Value is nth character of pname of x. Equivalent to car[nth[unpack[x];n]] but faster and does no conses. n can be negative, in which case counts from end of pname, e.g. -1 refers to last character, -2 the next to last, etc. If n is greater than the number of characters in the pname, or less than minus that number, or 0, value is NIL.
chcon[x;flg]	returns the pname of x as a list of (ASCII) character codes, i.e. numbers, e.g. chcon[FOO] = (70 79 79). If flg=t, the prin2-pname is used.
chcon1[x]	returns character code of first character of pname of x, e.g. chcon1[FOO] = 70. Thus chcon[x] = mapcar[unpack[x];chcon1]
dchcon[x;scratchlist;flg]	similar to dunpack
character[n]	n is an ASCII character *code*. Value is the atom having the corresponding single character as its pname* e.g. character[70] = F. Thus, unpack[x]=mapcar[chcon[x];character]

*See footnote p. 10.3.

gensym[]	Generates a new atom of the form Annnn, in which each of the n's is replaced by a digit. Thus, the first one generated is A0001, the second A0002, etc. This is a way of generating new atoms for various uses within the system. The value of gennum, initially 10000, determines the next gensym, e.g. if gennum is set to 10023, gensym[]=A0024.

The term gensym is also used to indicate an atom that was produced by the function gensym.

mapatoms[fn]	Applies fn to every literal atom in the system, e.g. mapatoms[(LAMBDA (X) (AND (SUBRP X) (PRINT X)))] will print every subr. Value is NIL.

String Functions

stringp[x]	Is _x_ if _x_ is a string, NIL otherwise. Note: if _x_ is a string, nlistp[x] is T, but atom[x] is NIL.
strequal[x;y]	Is _x_ if _x_ and _y_ are both strings and equal. _equal_ uses _strequal_. Note that strings may be _equal_ without being _eq_.
mkstring[x]	Value is string corresponding to _prinl_ of _x_.
rstring[]	Reads a string - see Section 14.
substring[x;n;m]	Value is substring of _x_ consisting of the _nth_ thru _mth_ characters of _x_. If _m_ is NIL, the substring is the nth character of _x_ thru the end of _x_. _n_ and _m_ can be negative numbers, a la _nthchar_, p. 10.4, i.e. equal[substring[x;1;-1];x] is T. Returns NIL if the substring is not well defined, e.g. _n_ or _m_ > nchars[x] or < minus[nchars[x]] or _n_ is to the right of _m_ in _x_. If _x_ is not a string, equivalent to substring[mkstring[x];n;m], except does not have to actually make a string if _x_ is a literal atom. (See next section on string storage).

gnc[x]	get next character of string x. Returns the next character of the string, (as an atom), *and* removes the character from the string. Returns NIL if x is the null string. If x isn't a string, a string is made. Used for sequential access to characters of a string. Note that if x is a substring of y gnc[x] does not remove the character from y, i.e. gnc doesn't physically change the string of characters, just the pointer and the byte count.*
glc[x]	gets last character of string x. Above remarks about gnc also apply to glc.
concat[x_1;x_2;...;x_n]	lambda nospread function. Concatenates (copies of) any number of strings. The arguments are transformed to strings if they aren't strings. Value is the new string, e.g. concat["ABC";DEF;"GHI"] = "ABCDEFGHI"
rplstring[x;n;y]	Replace characters of string x beginning at character n with string y. n may be positive or negative. x and y are converted to strings if they aren't already. Characters are smashed

*See string storage section that follows.

into (converted) x. Returns new x. Error if the new string would be longer than the original.* Note that if x is a substring of z, z will also be modified by the action of rplstring.

mkatom[x]

Creates an atom whose pname is the same as that of the string x or if x isn't a string, the same as that of mkstring[x], e.g. mkatom[(A B C)] is the atom %(A% B% C%). If atom would have > 99 characters, causes an error.

Searching Strings

strpos is a function for searching one string looking for another. Roughly it corresponds to member, except that it returns a character position number instead of a tail. This number can then be given to substring or utilized in other calls to strpos.

strpos[x;y;start;skip;anchor;tail]

x and y are both strings (or else they are converted automatically). Searches y beginning at character number start, (or else 1 if start is NIL) and looks for a sequence of characters equal to x. If a match is found, the corresponding character position is returned, otherwise NIL.

```
e.g.  strpos["ABC","XYZABCDEF"]=4
      strpos["ABC","XYZABCDEF";5]=NIL
      strpos["ABC","XYZABCDEFABC";5]=10
```

*If y was not a string, x will already have been partially modified since rplstring does not know whether y will 'fit' without actually attempting the transfer.

skip can be used to specify a character in x that matches any character in y, e.g.

strpos["A&C&";"XYZABCDEF";NIL;&]=4

If anchor is T, strpos compares x with the characters beginning at position start, or 1. If that comparison fails, strpos returns NIL without searching any further down y. Thus it can be used to compare one string with some *portion* of another string, e.g.

strpos["ABC";"XYZABCDEF";NIL;NIL;T]=NIL
strpos["ABC";"XYZABCDEF";4;NIL;T]=4

Finally, if tail is T, the value returned if successful is not the starting position of the sequence of characters corresponding to x, but the position of the first character after that, i.e. starting point plus nchars[x] e.g.

strpos["ABC";"XYZABCDEFABC";NIL;NIL;NIL;T]=7
Note that strpos["A";"A";NIL;NIL;NIL;T]=2

Example Problem

Given the strings x, y, and z, write a function foo that will make a string corresponding to that portion of x between y and z, e.g. foo["NOW IS THE TIME FOR ALL GOOD MEN" "IS" "FOR"] is " THE TIME ".

```
(FOO
  [LAMBDA (X Y Z)
    (AND (SETQ Y (STRPOS Y X NIL NIL NIL T))
         (SETQ Z (STRPOS Z X Y))
         (SUBSTRING X Y (SUB1 Z])
```

String Storage

A string is stored in 2 parts; the characters of the string, and a pointer to the characters. The pointer, or 'string pointer', indicates the *byte* at which the string begins and the length of the string. It occupies one word of storage. The characters of the string are stored in a portion of the LISP address space devoted exclusively to storing characters, five characters to a word.

Since the internal pname of literal atoms also consists of a pointer to the beginning of a string of characters and a byte count, conversion between literal atoms and strings does not require any additional storage for the characters of the pname, although one cell is required for the string pointer.*

When the conversion is done internally, e.g. as in substring or strpos, no additional storage is required for using literal atoms instead of strings.

The use of storage by the basic string functions is given below:

mkstring[x]	x string	no space
	x literal atom	new pointer
	other	new characters and pointer
substring[x;n;m]	x string	new pointer
	x literal atom	new pointer
	other	new characters and pointer

*Except when the string is to be smashed by rplstring. In this case, its characters must be copied to avoid smashing the pname of an atom. rplstring automatically performs this operation.

gnc[x];glc[x]	x string	no space, pointer is modified
	other	like <u>mkstring</u>, but doesn't make much sense
concat[x ...z]	args any type	new characters for whole new string, one new pointer
rplstring[x;n;y]	x string	no new space unless characters are in <u>pname</u> space (as result of mkstring[atom]) in which case <u>x</u> is quietly copied to string space
	x other	new pointer and characters
	y any type	type of y doesn't matter

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+2 words, of which the first two are header information. The next $p \le n$ are cells which will contain unboxed integers, and are initialized to unboxed Ø. The last $n-p \ge$ Ø cells will contain pointers initialized with v, i.e., both car and cdr are available for storing information, and each initially contain v. If p is NIL, Ø is used (i.e., an array containing all LISP pointers). The value of array is the array, also called an array pointer. If sufficient space is not available for the array, a garbage collection of array space, GC: 1, is initiated. If this is unsuccessful in obtaining sufficient space, an error is generated

Array-pointers print as #n, where n is the octal representation of the pointer. Note that #n will be read as an atom, and not an array pointer.

arraysize[a]	Returns the size of array a. Generates an error if a is not an array.
arrayp[x]	Value is x if x is an array pointer otherwise NIL. No check is made to ensure that x actually addresses the beginning of an array.

elt[a;n]	Value is nth element of the array a.* If n corresponds to the *unboxed* region of a, the value of elt is the full 36 bit word, as a boxed integer. If n corresponds to the *pointer* region of a, the value of elt is the car half of the corresponding element. elt generates an error if a is not the beginning of an array.**
seta[a;n;v]	sets the nth element of the array a. If n corresponds to the *unboxed* region of a, v *must* be a number, and is unboxed and stored as a full 36 bit word into the nth element of a. If n corresponds to the *pointer* region of a, v replaces the car half of the nth element.

Note that seta and elt are always inverse operations.

eltd[a;n]	same as elt for unboxed region of a, but returns cdr half of nth element, if n corresponds to the pointer region of a.
setd[a;n;v]	same as seta for unboxed region of a, but sets cdr half of nth element, if n corresponds to the pointer region of a.

In other words, eltd and setd are always inverse operations.

---

*actually corresponds to n+2nd cell because of the 2 word header.

**arrayp is true for pointers into the middle of arrays, but elt and seta must be given a pointer to the beginning of an array, i.e., a value of array.

Storage Functions

reclaim[n]	Initiates a garbage collection of type _n_. Value of _reclaim_ is number of words available (for that type) after the collection.

Garbage collections, whether invoked directly by the user or indirectly by need for storage, do not confine their activity solely to the data type for which they were called, but automatically collect some or all of the other types.

ntyp[x]	Value is type number for the data type of LISP pointer _x_, e.g. ntyp[(A . B)] is 8, the type number for lists. Thus GC: 8 indicates a garbage collection of list words.

type	number
arrays, compiled code	1
stack positions	2
list words	8
atoms	12
floating point numbers	16
large integers	18
small integers	20
string pointers	24
pname storage	28
string storage	30

typep[x;n]	eq[ntyp[x];n]
gcgag[message]	_message_ is a string or atom to be printed (using _prin1_) wherever a garbage collection is begun. If _message_=T, its standard setting, GC: is printed, followed by the type number. When the garbage collection is complete, two numbers will be printed out: the number of words collected for that type, and the total number of words available for that type, i.e. allocated but not necessarily currently in use (see _minfs_ below).

Example:

```
←RECLAIM(18)

GC: 18
511, 3071 FREE WORDS
3071
←RECLAIM(12)

GC: 12
1020, 1020 FREE WORDS
1020
```

If message=NIL, no garbage collection message is printed, either on entering or leaving the garbage collector. Value of gcgag is old setting.

minfs[n;typ]

Sets the minimum amount of free storage which will be maintained by the garbage collector for data types of type number typ. If, after any garbage collection for that type, fewer than n free words are present, sufficient storage will be added (in 512 word chunks) to raise the level to n. If n=NIL, 8 is used, i.e. minfs refers to list words.

A minfs setting can also be changed dynamically, even during a garbage collection, by typing control-S followed by a number, followed by a period.*

---

*When the control-S is typed, LISP immediately clears and saves the input buffer, rings the bell, and waits for input, which is terminated by any non-number. The input buffer is then restored and the program continues. If the input was terminated by other than a period, the whole interaction is ignored.

If the control-S was typed during a garbage collection, the number is the new <u>minfs</u> setting for the type being collected, otherwise for type 8, i.e. list words.

Note: A garbage collection of a 'related' type may also cause more storage to be assigned to that type. See discussion of garbage collector algorithm, Section 3.

storage[flg]

Prints amount of storage (by type number) used by and assigned to the user, e.g.

```
←STORAGE]

TYPE USED       ASSIGNED
1    80072      87552
8    7970       9216
12   7032       7680
16   0          512
18   1124       2560
24   118        512
28   4226       4608
30   573        1024
SUM  101115     113664
```

If <u>flg</u>=T, includes storage used by and assigned to the system. Value is NIL.

gctrp[n]	garbage collection trap. Causes a (simulated) control-H interrupt when the number of free list words (type 8) remaining equals n, i.e. when a garbage collection would occur in n more conses. The message GCTRP is printed, the function interrupt (Section 16) is called, and a break occurs. Note that by advising (Section 19) interrupt the user can program the handling of a gctrp instead of going into a break. Value of gctrp is its last setting. gctrp[-1] will 'disable' a previous gctrp since there are never -1 free list words. gctrp is initialized this way. gctrp[] is number of list words left, i.e. number of conses until next type 8 garbage collection, see p. 21.4.
conscount[]	Value is number of conses since LISP started up. If given a number, resets conscount to that number.
closer[a;x]	Stores x into memory location a. Both x and a must be numbers.
openr[a]	Value is number in memory location a, i.e. boxed.

SECTION XI

FUNCTIONS WITH FUNCTIONAL ARGUMENTS

Contents

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 freely in a functional argument. All system functions standardly use variable names consisting of the function name concatenated with x or fn etc. However, by specifying the free variables used in a functional argument as the second argument to function, thereby using the BBN-LISP FUNARG feature, the user can be sure of no clash.

function[x;y]

is an nlambda function. If y=NIL, the value of function is x, i.e., function is identical to quote, for example (MAPC LST (FUNCTION PRINT)) will cause mapc to be called with two arguments, the value of lst and PRINT. Similarly,

```
(MAPCAR LST (FUNCTION (LAMBDA (Z)
        (LIST (CAR Z)))))
```

will cause mapcar to be called with the value of lst and

(LAMBDA (Z) (LIST CAR Z))). When compiled, function will cause code to be compiled for x; quote will not. Thus (MAPCAR LST (QUOTE (LAMBDA --))) will cause mapcar to be called with the value of lst and the expression (LAMBDA --). The functional argument will therefore still be interpreted. The corresponding expression using function will cause a dummy function to be created with definition (LAMBDA --), and then compiled. mapcar would then be called with the value of lst and the name of the dummy function. See p. 18.16.

If y is not NIL, it is a list of variables that are (presumably) used freely by x. In this case, the value of function is an expression of the form (FUNARG x array), where array contains the variable bindings for those variables on y. Funarg is described on pp. 11.6-11.7.

map[mapx;mapfnl;mapfn2]

If mapfn2 is NIL this function applies the function mapfnl to successive tails of the list mapx. That is, first it computes mapfnl[mapx], and then mapfnl[cdr[mapx]], etc., until

	mapx is exhausted.* If *mapfn2* is provided, mapfn2[mapx] is used instead of cdr[mapx] for the next call for *mapfnl*, e.g., if *mapfn2* were *cddr*, alternate elements of the list would be skipped. The value of *map* is NIL.
mapc[mapx;mapfnl;mapfn2]	Identical to *map*, except that mapfnl[car[mapx]] is computed each time instead of mapfnl[mapx], i.e., *mapc* works on elements, *map* on tails. The value of *mapc* is NIL.
maplist[mapx;mapfnl;mapfn2]	computes successively the same values that *map* would compute; and returns a list consisting of those successive values.
mapcar[mapx;mapfnl;mapfn2]	computes the same values that *mapc* would compute, and returns a list consisting of those values. e.g. mapcar[x;FNTYP] is a list of *fntyps* for each element on *x*.

---

*i.e., becomes a non-list.

mapcon[mapx;mapfn1;mapfn2] Computes the same values as map and maplist, but nconcs these values to form a list which it returns.

mapconc[mapx;mapfn1;mapfn2] Computes the same values as mapc and mapcar, but nconcs the values to form a list which it returns.

Note that mapcar creates a new list which is a mapping of the old list in that each element of the new list is the result of applying a function to the corresponding element on the original list. mapconc is used when there are a *variable* number of elements (including none) to be inserted at each iteration. e.g. mapconc[X;(LAMBDA (Y) (AND Y (LIST Y)))] will make a list consisting of x with all NILs removed, mapconc[X;(LAMBDA (Y) (AND (LISTP Y) Y))] will make a linear list consisting of all the lists on x, e.g. it applied to ((A B) C (D E F) (G) H I) will yield (A B D E F G).*

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 when *either* mapx or mapy are exhausted.

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 when either mapx or mapy is exhausted.

*Note that since mapconc uses nconc to string the corresponding lists together, in this example, the original list will be clobbered, i.e. it would now be ((A B D E F G) C (D E F G) (G) H I). If this is an undesirable side effect, the functional argument to mapconc should return instead a top level copy, e.g. in this case, use (AND (LISTP Y) (APPEND Y))).

**mapfn2 is still a function of one argument, and is applied twice on each iteration; mapfn2[mapx] gives the new mapx, mapfn2[mapy] the new mapy. cdr is used if mapfn2 is not supplied, i.e., is NIL.

maprint[lst;file;left;right;sep;pfn]	is a general printing function. It cycles through lst applying pfn (or prinl if pfn not given) to each element of lst Between each application it performs prinl of sep, or " " if not given. If left is given, it is printed (using prinl) initially; if right is given it is printed (using prinl) at the end. For example, maprint[x;NIL;%(;%)] is equivalent to prinl for lists. To print a list with commas between each element and a final '.' one could use maprint[x;T;NIL;%.;%,].
mapdl,searchpdl	See Section 12.
mapatoms	See Section 5.
every,some,notevery,notany	See Section 5.

Funarg

function is a function of two arguments, x, a function, and y a list of variables used freely by x. If y is not NIL, the *value* of function is an expression of the form (FUNARG x array), where array contains the bindings of the variables on y at the time the call to function was evaluated. funarg is not a function itself. Like LAMBDA and NLAMBDA, it has meaning and is specially recognized by LISP only in the context of applying a function to arguments. In other words, the expression (FUNARG x array) is used exactly like a function.* When a funarg is applied, the stack is modified so that the bindings contained in the array will be in force when x, the function, is called.**

For example, suppose a program wished to compute (FOO X (FUNCTION FIE)), and fie used y and z as free variables. If foo rebound y and z, fie would obtain the rebound values when it was called from inside of foo. By evaluating instead (FOO X (FUNCTION FIE (Y Z))), foo would be called with (FUNARG FIE array) as its second argument, where array contained the bindings of y and z (at the time foo was called). Thus when fie was called from inside of foo, it would 'see' the original values of y and z.

However, funarg is more than just a way of circumventing the clashing of variables. For example, a funarg expression can be returned as the value of a computation, and then used 'higher up', e.g., when the bindings of the variables contained in array were no longer on the stack. Furthermore, if the function in a

* LAMBDA, NLAMBDA, and FUNARG expressions are sometimes called 'function objects' to distinguish them from functions, i.e., literal atoms which have function definitions.

**The implementation of funarg is described on pp. 12.13-12.14.

funarg expression *sets* any of the variables contained in the array, the array itself (and only the array) will be changed. For example, suppose foo is defined as

```
   (LAMBDA (LST FN) (PROG (Y Z) (SETQ Y &) (SETQ Z &)
               ... (MAPC LIST FN) ...))
```

and (FOO X (FUNCTION FIE (Y Z))) is evaluated. If one application of fie (by the mapc in foo) changes y and z, then the next application of fie will obtain the changed values of y and z resulting from the previous application of fie, since both applications of fie come from the exact same funarg object, and hence use the exact same array. The bindings of y and z bound inside of foo, and the bindings of y and z *above* foo would not be affected. In other words, the variable bindings contained in array are a part of the function object, i.e., the funarg carries its environment with it.

Thus by creating a funarg expression with function, a program can create a function object which has updateable binding(s) associated with that object which last *between* calls to it, but are only accessible through that instance of the function. For example, using the funarg device, a program could maintain two different instances of the same random number generator in different states, and run them independently.

Example

If foo is defined as (LAMBDA (X) (COND ((ZEROP A) X) (T (MINUS X)))) and fie as (LAMBDA NIL (PROG (A) (SETQ A 2) (RETURN (FUNCTION FOO)))). then if we perform (SETQ A Ø), (SETQ FUM (FIE)), the value of fum is FOO, and the value of (FUM 3) is 3, because the value of A at the time foo is called is Ø.

However if fie were defined instead as (LAMBDA NIL (PROG (A) (SETQ A 2) (RETURN (FUNCTION FOO (A))))), the value of fum would be (FUNARG FOO array) and so the value of (FUM 3) would be -3, because the value of A seen by foo is the value A had when the funarg was created inside of fie, i.e. 2.

SECTION XII

VARIABLE BINDINGS AND PUSH DOWN LIST FUNCTIONS

Contents

A number of schemes have been used in different implementations 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 a pushdown list, and restoring these values when exiting from a function.

4. Storing values on a 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 possibly 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 also be kept. The pushdown list thus contains pairs, each consisting of a variable name and its value. Each pair occupies one word or 'slot' on the pushdown list, with the name in the left half, i.e. <u>cdr</u>, and the value in the right half, i.e. <u>car</u>. The interpreter gets the value of a variable by searching back up the pushdown list looking for a 'slot' for which <u>cdr</u> is the name of the variable. <u>car</u> is then its value.

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 to find the value of a variable. 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 make possible 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.

* Also called the stack.

There are three pushdown lists used in BBN 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; and the third is called the number stack and is used for storing temporary partial results of numeric operations.

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 pushdown lists in the actual implementation is for efficiency 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 (intentionally faulty):

```
(FACT
  [LAMBDA (N)
    (COND
      ((ZEROP N)
        L)
      (T (ITIMES N (FACT (SUB1 N])
```

In evaluating (FACT 1) 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 true, 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 *itimes*. However before *itimes* can be called, its arguments must be evaluated. The first argument is evaluated by searching the stack for the last binding of *n*; 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 has been evaluated, and another is awaiting evaluation. The output below illustrates this by showing the state of the push-down list at the point in the computation of (FACT 1) when the unbound atom L is reached.

```
←FACT(1)
U.B.A.
(L BROKEN)
:BTV!

   *FORM* (BREAK1 L T L NIL #41050)
   FAULTX L
   #0 (L)

   #0 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))              1
COND

   *FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
   #0 ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))       2

   N 0
FACT

   *FORM* (FACT (SUB1 N))
   #2 ITIMES
   #0 ((FACT (SUB1 N)))                                           3
   #0 1                                                           4
   *FORM* (ITIMES N (FACT (SUB1 N)))
   #0 ((ITIMES N (FACT (SUB1 N))))                                5

   #0 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))              6
COND

   *FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
   #0 ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))       7

   N 1
FACT

**TOP**
```

Internal calls to eval, e.g., from cond and the interpreter, are marked on the push-down list by a special mark called an eval-blip. eval-blips are indicated by the appearance of (VAG 16) in the left-half, i.e. the variable name position, for that slot. They are printed by the backtrace as *FORM*. The genealogy of *FORM*'s is thus a history of the computation. Other temporary information is frequently recorded on the push-down list in slots for which the 'variable name' is (VAG Ø), which prints as #Ø. In this example, this information consists of (1) the tail of a list of cond clauses, (2) the tail of an implicit progn, i.e., the definition of fact, (3) the tail of an argument list, (4) the value of a previously evaluated argument, (5) the tail of a cond clause whose predicate evaluated to true, and (6) and (7) same as (1) and (2).

Note that a function is not actually entered and does not appear on the stack, until its arguments have been evaluated.* Also note that the #0 'bindings' comprise the actual working storage. In other words, in the above example, if a (lower) function changed the value of the binding at (1) the cond would continue interpreting the new binding as a list of cond clauses. Similarly, if (4) were changed, the new value would be given to itimes as its first argument after its second argument had been evaluated, and itimes was actually called.

*except for functions which do not have their arguments evaluated, although they themselves may call eval, e.g. cond.

The Pushdown List and Compiled Functions

Calls to compiled functions, and the bindings of their arguments, i.e. names and values, are handled in the same way as for interpreted functions (hence the compatibility between interpreted and compiled functions). However, compiled functions treat free variables in a special way that interpreted functions do not. Interpreted functions 'look up' free variables when 'they get to them,' and may look up the same variable many times. However, compiled functions look up each free variable only once.* Whenever a compiled function is entered, the pushdown list is scanned and the most recent binding for each free variable used in the function is found (or value cell if no binding) and stored in the right half of a slot on the stack (an unboxed 0 is stored in the left half to distinguish this 'binding' from ordinary bindings). Thus, following the bindings of their arguments, compiled functions store on the pushdown list pointers to the bindings for each free variable used in the function.

In addition to the pointers to free variable bindings, compiled functions differ from interpreted functions in the way they treat locally bound variables, i.e. <u>progs</u> and open <u>lambdas</u>. Whereas in interpreted functions <u>progs</u> and open <u>lambdas</u> are called in the ordinary way as functions, in compilation, <u>progs</u> and open <u>lambdas</u> disappear, although the variables bound by them are stored on the stack in the conventional manner so that functions called from inside them can reference the variables. These variables appear on the stack following the arguments to the compiled function (if any) and the free variable pointers (if any). The only way to determine dynamically what variables are bound locally by a compiled function is to search the stack from the first slot *beyond* the *last* argument to the function (which can be found with <u>stknargs</u> and <u>stkarg</u> described below), to the slot corresponding to the *first* argument of the *next* function. Any slots encountered that contain literal atoms in their left half are local bindings.

*A list of all free variables is generated at compile time, and is in fact computable from the compiled definition. See Chapter 18.

Pushdown List Functions

NOTE: Unless otherwise stated, for all pushdown list functions, pos is a position on the control stack. If pos is a literal atom other than NIL, (STKPOS pos 1) is used. In this case, if pos is not found, i.e., stkpos returns NIL, and ILLEGAL STACK ARG error is generated.

stkpos[fn;n;pos]	Searches the control stack starting at pos for the nth occurrence of fn. Returns control stack position of that fn if found,* else NIL. If n is positive, searches backward (normal usage). If n is negative, searches forward, i.e., down the control stack. For example, stkpos[FOO;-2;FIE] finds second call to FOO after (below) the last call to FIE. If n is NIL, 1 is user. If pos is NIL, the search starts at the current position. stkpos[] is the current position.
stknth[n;pos]	Value is the stack position (control stack) of the nth function call relative to position pos. If pos is NIL, the top of stack is assumed for n>0, and the current position is assumed for n<0. I.e., stknth[-1] is the call before stknth, stknth[1] is the call to evalqt at the top level. Value of stknth is NIL if there is no such call - e.g., stknth[10000] or stknth[-10; stknth[5]].

* A stack position is a pointer to the corresponding slot on the control or parameter stack, i.e., the address of that cell. It prints as an unboxed number, e.g., #32002, and its type is 2 (Section 10).

stkname[pos]	Value is the name of the function at control stack position *pos*. In this case, *pos* must be a real stack position, not an atom.

Thus *stkpos* converts function names to stack positions, *stknth* converts numbers to stack positions, and *stkname* converts positions to function names.

Information about the variables bound at a particular function call can be obtained using the following functions:

stknargs[pos]	Value is the number of arguments bound by the function at position *pos*.
stkarg[n;pos]	Value is a pointer to the *n*th argument (named or not)* of the function at position *pos*, i.e., the value is a parameter stack position. *car* of this pointer gives the value of the binding, *cdr* the *name*. *n*=1 corresponds to the first argument at *pos*. *n* can be Ø or negative, i.e., stkarg[Ø;FOO] is a pointer to the slot immediately before the first argument to FOO, stkarg[-1;FOO] the one before that, etc.

Note that the user can change (set) the value of a particular binding by performing an *rplaca* on the value of *stkarg*. Similarly, *rplacd* changes (sets) the name.

---

*Subrs do not store the names of their arguments.

The value of stkarg is a position (slot) on the parameter stack. There is currently no analogue to stknth for the parameter stack. However, the parameter stack is a contiguous block of memory, so to obtain the slot previous to a given slot, perform vag[subl[loc[slot]]]; to obtain the next slot perform vag[addl[loc[slot]]], i.e. stkarg[2;pos] =
vag[addl[loc[stkarg[1;pos]]]].*

As an example of the use of stknargs and stkarg:

variables[pos]	returns list of variables bound at pos.

can be defined by

```
(VARIABLES
  [LAMBDA (POS)
    (PROG (N L)
          (SETQ N (STKNARGS POS))
      LP  (COND
            ((ZEROP N)
              (RETURN L)))
          (SETQ L (CONS (CDR (STKARG N POS))
                        L))
          (SETQ N (SUB1 N))
          (GO LP])
```

The counterpart of variables is also available.

stkargs[pos]	Returns list of values of variables bound at pos.

*See Section 13 for discussion of vag and loc.

The next three functions, stkscan, evalv, and stkeval all involve searching the parameter pushdown stack. For all three functions, pos may be a position on the control stack, i.e., a value of stkpos or stknth.* In this case, the search starts at stkarg[stknargs[pos];pos], i.e., it will include the arguments to the function at pos but not any locally bound variables. pos may also be a position on the *parameter* stack, in which case the search starts with, and includes that position. Finally, pos can be NIL, in which case the search starts with the current position on the parameter stack.

stkscan[var;pos]	Searches backward on the parameter stack from pos for a binding of var. Value is the slot for that binding if found, i.e., a parameter stack position, otherwise var itself (so that car of stkscan is always the value of var).
evalv[var;pos]	car[stkscan[var;pos]], i.e., returns the value of the atom var as of position pos.
stkeval[pos;form]	is a more general evalv. It is equivalent to eval[form] at position pos, i.e., all *variables* evaluated in form, will be evaluated as of pos.**

* or a function name, which is equivalent to stkpos[pos;1] as described earlier.

**However, any functions in form that specifically reference the stack, e.g., stkpos, stknth, retfrom, etc., 'see' the stack as it currently is. (See pp. 12.13, 12.14 for description of how stkeval is implemented.)

Finally, we have two functions which clear the stacks:

retfrom[pos;value]	clears the stack back to the function at position pos, and effects a return from that function with value as its value.
reteval[pos;form]	clears the stack back to the function at position pos, *then* evaluates form and returns with its value to next higher function. I.e., reteval[pos,form] = retfrom[pos;stkeval[pos;form]],if form does not involve any stack functions itself.

We also have:

mapdl[mapdlfn;mapdlpos]	starts at position mapdlpos (current if NIL), and applies mapdlfn to the function name at each pushdown position, i.e., to stkname[mapdlpos] until the top of stack is reached. Value is NIL. mapdlpos is updated at each iteration.

For example,
```
mapdl[(LAMBDA (X) (COND ((EQ (FNTYP X) (QUOTE EXPR)) (PRINT X]
```
will print all exprs on the push-down list.
```
mapdl [(LAMBDA (X) (COND ((GREATERP (STKNARG MAPDLPOS) 2) (PRINT X]
```
will print all functions of more than two arguments.

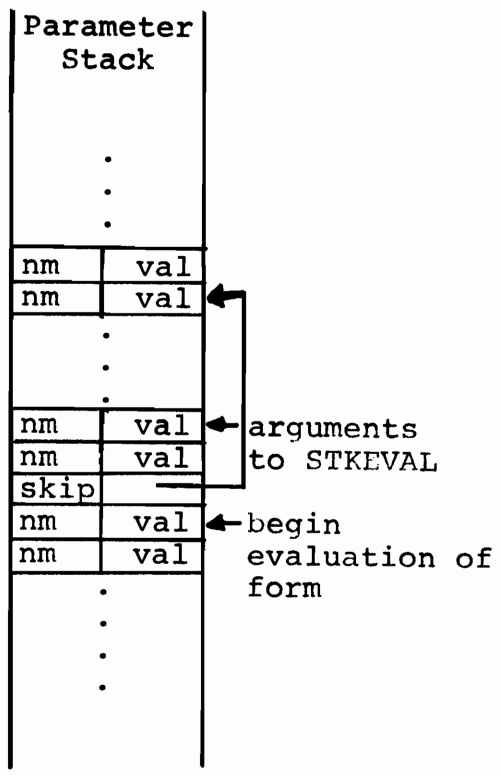
searchpdl[srchfn;srchpos]	searches the pushdown list starting at position srchpos (current if NIL) until it finds a position for which srchfn applied to the name of the function called at that position is not NIL. Value is (name . position) if such a position is found, otherwise NIL. srchpos is updated at each iteration.

The Pushdown List and Funarg

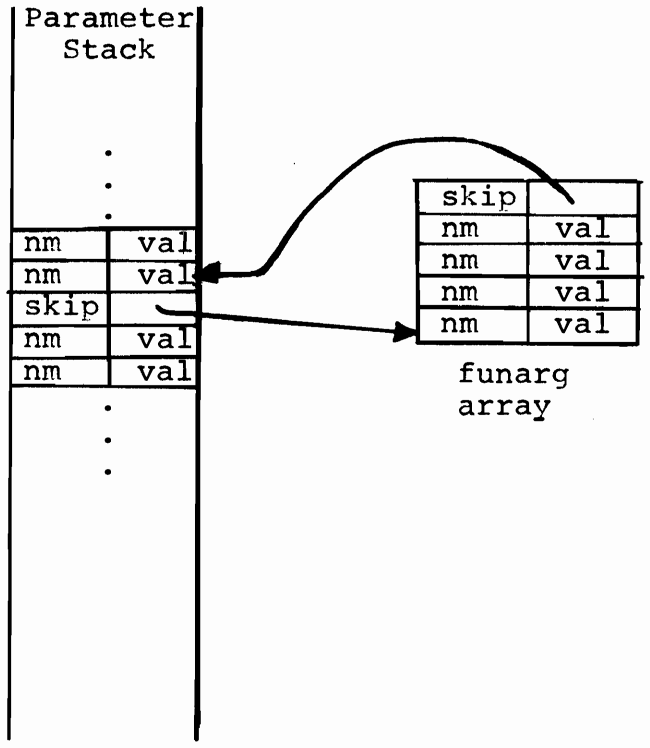
The linear scan up the parameter stack for a variable binding can be interrupted by a special mark called a skipblip appearing on the stack in a name position (See figure on p. 12.14). In the value position is a pointer to the position on the stack where the search is to be continued. This is what is used to make stkeval, p. 12.11 work. It is also used by the funarg device (p. 11.6).

When a funarg is applied, LISP puts a skipblip on the parameter stack with a pointer to the funarg array, and another skipblip at the top of the funarg array pointing back to the stack. The effect is to make the stack look like it has a patch. The names and values stored in the funarg array will thus be seen before those higher on the stack. Similarly, setting a variable whose binding is contained in the funarg array will change only the array. Note however that as a consequence of this implementation, *the same instance of a funarg object cannot be used recursively.*

Use of 'SKIPBLIPs'



STKEVAL



FUNARG

SECTION XIII

NUMBERS AND ARITHMETIC FUNCTIONS

Contents

General Comments

There are three different types of numbers in BBN LISP: small integers, large integers, and floating point numbers.* Since a large integer or floating point number can be (in value) any 36 bit quantity (and vice versa), it is necessary to distinguish between those 36 bit quantities that represent large integers or floating point numbers, and other LISP pointers. We do this by "boxing" the number, which is sort of like a special "cons": when a large integer or floating point number is created (via an arithmetic operation or by read), LISP gets a new word from "number storage" and puts the large integer or floating point number into that word. LISP then passes around the pointer to that word, i.e., the "boxed number", rather than the actual 36 bit quantity itself. Then when a numeric function needs the actual numeric quantity, it performs the extra level

* Floating point numbers are created by the read program when a . or an E appears in a number, e.g., 1000 is an integer, 1000. a floating point number, as are 1E3 and 1.E3. Note that 1000D, 1000F, and 1E3D are perfectly legal literal atoms.

of addressing to obtain the 'value' of the number. This latter process is called "unboxing". Note that unboxing does not use any storage, but that each boxing operation uses one new word of number storage. Thus, if a computation creates many large integers or floating point numbers, i.e., does lots of boxes, it may cause a garbage collection of large integer space, GC:18, or of floating point number space, GC:16.

Small Integers

Small integers are those integers for which smallp is true, currently integers whose absolute value is less than 1536. Small integers are boxed by offsetting them by a constant so that they overlay an area of LISP's address space that does not correspond to any LISP data type. Thus boxing small numbers does not use any storage, and furthermore, each small number has a unique representation, so that eq may be used to check equality. Note that eq should not be used for large integers or floating point numbers, e.g., eq[2000;add1[1999]] is NIL! eqp or equal must be used instead.

Integer Arithmetic

All of the functions described below work on integers. Unless specified otherwise, if given a floating point number, they first convert the number to an integer by truncating the fractional bits, e.g., iplus[2.3,3.8]=5; if given a non-numeric argument, they generate an error.

It is important to use the *integer* arithmetic functions, whenever possible, in place of the more general arithmetic functions which allow mixed floating point and integer arithmetic, e.g., iplus vs plus, igreaterp vs greaterp, because the integer functions compile open, and therefore run faster than the general arithmetic functions, and because the compiler is "smart" about eliminating unnecessary boxing and unboxing. Thus, the expression

```
    (IPLUS (IQUOTIENT (ITIMES N 100) M) (ITIMES X Y))
```

will compile to perform only one box, the outer one, and the expression

```
    (IGREATERP (IPLUS X Y) (IDIFFERENCE A B))
```

will compile to do no boxing at all.

Note that the PDP-10 is a 36 bit machine, so that all integers are between -2^{35} and $2^{35}-1$.* Adding two integers which produce a result outside this range causes overflow, e.g., $2^{34}+2^{34}$.

The procedure on overflow is to return the largest possible integer, i.e. $2^{35}-1$ or else generate an error.** The function overflow dictates the choice:

overflow[] - return a value, overflow[T] - give an error.
overflow[] is the standard setting.

*Approximately 34 billion

**If the overflow occurs by trying to create a negative number of too large a magnitude, -2^{35} is used instead of $2^{35}-1$.

Integer Functions

iplus[x_1;x_2;...;x_n]	x_1+x_2+...+x_n
iminus[x]	- x
idifference[x;y]	x - y
add1[x]	x + 1
sub1[x]	x - 1
itimes[x_1;x_2;...;x_n]	the product of x_1,x_2,...x_n
iquotient[x;y]	x/y truncated, e.g., iquotient[3;2]=1, iquotient[-3,2]=-1
iremainder[x;y]	the remainder when x is divided by y, e.g., iremainder[3;2]=1
igreaterp[x;y]	T if x>y; NIL otherwise
ilessp[x;y]	T is x<y; NIL otherwise
zerop[x]	defined as eq[x;Ø]. Note that zerop should not be used for floating point numbers because it uses eq. Use eqp[x;Ø] instead.

minusp[x]	T if *x* is negative; NIL otherwise. Does not convert *x* to an integer, but simply checks sign bit.
eqp[n;m]	T if *n* and *m* are *eq*, or equal numbers, NIL otherwise. (*eq* may be used if *n* and *m* are known to be small integers.) *eqp* does not convert *n* and *m* to integers, e.g., eqp[2000;2000.3]=NIL, but it can be used to compare an integer and a floating point number, e.g., eqp[2000;2000.0]=T. *eqp* does *not* generate an error if *n* or *m* are not numbers.
smallp[n]	T if *n* is a small integer, else NIL. *smallp* does *not* generate an error if *n* is not a number.
fixp[x]	*x* if *x* is an integer, else NIL. Does *not* generate an error if *x* is not a number.
fix[x]	Converts *x* to an integer by truncating fractional bits, e.g., fix[2.3] = 2, fix[-1.7] = -1. If *x* is already an integer, fix[x]=x and doesn't use any storage.

logand[x_1;x_2;...;x_n]	lambda no-spread, value is logical and of all its arguments, as an integer, e.g., logand[7;5;6]=4.
logor[x_1;x_2;...;x_n]	lambda no-spread, value is the logical or of all its arguments, as an integer, e.g., logor[1;3;9]=11.
logxor[x_1;x_2;...;x_n]	lambda no-spread, value is the logical exclusive or of its arguments, as an integer, e.g., logxor[11;5]=14, logxor[11;5;9] = logxor[14;9]=7.
lsh[n;m]	(arithmetic) left shift, value is $n*2^m$, i.e., n is shifted left m places. n can be positive or negative. If m is negative, n is shifted *right* -m places.
rsh[n;m]	(arithmetic) right shift, value is $n*2^{-m}$, i.e., n is shifted right m places. n can be positive or negative. If m is negative, n is shifted *left* -m places.
llsh[n;m]	logical left shift. On PDP-10, llsh is equivalent to lsh.

lrsh[n;m]   logical right shift.

The difference between a logical and arithmetic right shift lies in the treatment of the sign bit for negative numbers. For arithmetic right shifting of negative numbers, the sign bit is propagated, i.e., the value is a negative number. For logical right shift, zeroes are propagated. Note that shifting (arithmetic) a negative number 'all the way' to the right yields -1, not Ø.

Floating Point Arithmetic

All of the functions described below work on floating point numbers. Unless specified otherwise, if given an integer, they first convert the number to a floating point number, e.g., fplus[1;2.3] = fplus[1.0;2.3] = 3.3; if given a non-numeric argument, they generate an error.

The largest floating point number is 1.7014118E38, the smallest positive (non-zero) floating point number is 1.4693679E-39. The procedure on overflow is the same as for integer arithmetic, and the function overflow has the same effect. For underflow, i.e. trying to create a number of too small a magnitude, the value will be ∅ (if a value is to be returned).

fplus[x_1;x_2;...x_n]	$x_1+x_2...+x_n$
fminus[x]	- x
ftimes[x_1;x_2;...;x_n]	$x_1*x_2...*x_n$
fquotient[x;y]	x/y
fremainder[x;y]	the remainder when x is divided by y, e.g., fremainder[1.0;3.0]= 3.72529E-9.
minusp[x]	T if x is negative; NIL otherwise. Works for both integers and floating point numbers.
eqp[x;y]	T if x and y are eq, or equal numbers. See discussion p. 13.5.
fgtp[x;y]	T if x>y, NIL otherwise.

floatp[x]	is x if x is a floating point number; NIL otherwise. Does *not* give an error if x is not a number.

Note that if numberp[x] is true, then either fixp[x] or floatp[x] is true.

float[x]	Converts x to a floating point number, e.g., float[Ø] = Ø.Ø.

General Arithmetic

The functions in this section are 'contagious floating point arithmetic' functions, i.e., if any of the arguments are floating point numbers, they act exactly like floating point functions, and float all arguments and return a floating point number as their value. Otherwise, they act like the integer functions. If given a non-numeric argument, they generate an error.

plus[x_1;x_2;...;x_n]	$x_1+x_2+...+x_n$
minus[x]	$- x$
difference[x;y]	$x - y$
times[x_1;x_2;...;x_n]	$x_1*x_2*...*x_n$
quotient[x;y]	if *x* and *y* are both integers, value is iquotient[x;y], otherwise fquotient[x;y].
remainder[x;y]	if *x* and *y* are both integers, value is iremainder[x;y], otherwise fremainder[x;y].
greaterp[x;y]	T if x>y, NIL otherwise.
lessp[x;y]	T if x<y, NIL otherwise.
abs[x]	*x* if x>Ø, otherwise -x. *abs* uses *greaterp* and *minus*, (not *igreaterp* and *iminus*).

Special Functions

These functions are all "borrowed" from the FORTRAN library and handcoded in LISP via ASSEMBLE. They utilize a power series expansion and their values are (supposed to be) 27 bits accurate, e.g., sin[30]=.5 exactly.

expt[m;n]	value is m^n. If _m_ is an integer and _n_ is a positive integer, value is an integer, e.g., expt[3;4]=81, otherwise the value is a floating point number. If _m_ is negative and _n_ fractional, an error is generated.
sqrt[n]	value is a square root of _n_ as a floating point number. _n_ may be fixed or floating point. Generates an error if _n_ is negative. sqrt[n] is about twice as fast as expt[n;.5]
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, e.g., antilog[1] = e = 2.71828...
sin[x;radiansflg]	_x_ in degrees unless _radiansflg_=T. Value is sine of _x_ as a floating point number.
cos[x;radiansflg]	Similar to _sin_.
tan[x;radiansflg]	Similar to _sin_.

arcsin[x;radiansflg]

x is a number between -1 and 1 (or an error is generated). The value of arcsin is a floating point number, and is in degrees unless radiansflg=T. In other words, if arcsin[x;radiansflg]=z then sin[z;radiansflg]=x. The range of the value of arcsin is -90 to +90 for degrees, $-\frac{\pi}{2}$ to $+\frac{\pi}{2}$ for radians.

arcos[x;radiansflg]

Similar to arcsin. Range is Ø to 180, Ø to π.

arctan[x;radiansflg]

Similar to arcsin. Range is Ø to 180, Ø to π.

rand[lower;upper]

Value is a pseudo-random number between lower and upper inclusive, i.e. rand can be used to generate a sequence of random numbers. If both limits are integers, the value of rand is an integer, otherwise it is a floating point number. The algorithm is completely deterministic, i.e. given the same initial state, rand produces the same sequence of values. The internal state of rand is initialized using the function randset described below, and is stored on the free variable randstate.

randset[x]

Value is internal state of rand after randset has finished operating, (as a dotted pair of two integers). If x=NIL, no changes are made, i.e. value is current state. If x=T, randstate is initialized using the clocks. Otherwise, x is interpreted as a previous internal state, i.e. a value of randset, and is used to reset randstate. For example,

1. (SETQ OLDSTATE (RANDSET))
2. Use rand to generate some random numbers.
3. (RANDSET OLDSTATE)
4. rand will generate same sequence as in 2.

Reusing Boxed Numbers - setn

rplaca and rplacd provide a way of cannibalizing list structure for reuse in order to avoid making new structure and causing garbage collections.* This section describes an analogous function for large integers and floating point numbers, setn. setn is used like setq, i.e., its first argument is considered as quoted, its second is evaluated. If the current value of the variable being set is a large integer or floating point number, the new value is deposited into that word in number storage, i.e., no new storage is used.** If the current value is *not* a large integer or floating point number, e.g., it can be NIL, setn operates exactly like setq, i.e., the large integer or floating point number is boxed, and the variable is set. This eliminates initialization of the variable.

setn will work interpretively, i.e., reuse a word in number storage, but will not yield any savings of storage because the boxing (of the second argument) has already taken place, i.e., *before* setn was called. The elimination of a box is achieved only when the call to setn is compiled, since setn compiles open, and does not perform the box if the old value of the variable can be reused.

Caveats

There are three situations to watch out for when using setn. The first occurs when the same variable is being used for floating point numbers and large integers. If the current value

* This technique is frowned upon except in well-defined, localized situations where efficiency is paramount.

**The second argument to setn must always be a number or an error is generated.

of the variable is a floating point number, and it is reset to a large integer, via setn, the large integer is simply deposited into a word in floating point number storage, and hence will be interpreted as a floating point number. Thus,

```
←(SETQ FOO 2.3)
2.3
←(SETN FOO 10000)
2.189529E-43
```

Similarly, if the current value is a large integer, and the new value is a floating point number, equally strange results occur.

The second situation occurs when a setn variable is reset from a large integer to a small integer. In this case, the small integer is simply deposited into large integer storage. It will then print correctly, and function arithmetically correctly, but it is *not* a small integer, and hence not eq to another integer of the same value, e.g.,

```
←(SETQ FOO 10000)
10000
←(SETN FOO 1)
1
←(IPLUS FOO 5)
6
←(EQ FOO 1)
NIL
←(SMALLP FOO)
NIL
```

In particular, note that zerop will return NIL even if the variable is equal to Ø. Thus a program which begins with FOO set to a large integer and counts it down by (SETN FOO (SUB1 FOO)) must terminate with (EQP FOO Ø), not (ZEROP FOO).

Finally, the third situation to watch out for occurs when you want to save the current value of a setn variable for later use. For example, if FOO is being used by setn, and the user wants to save its current value on FIE, (SETQ FOO FIE) is not sufficent, since the next setn on FOO will also change FIE, because it changes the word in number storage pointed to by FOO, and hence pointed to by FIE. The number must be copied, e.g., (SETQ FIE (IPLUS FOO)), which sets FIE to a new word in number storage.

setn[var;x]	nlambda function like setq. var is quoted, x is evaluated, and its value must be a number. var will be set to this number. If the current value of var is a large integer or floating point number, that word in number storage is cannibalized. The value of setn is the (new) value of var.

Box and Unbox

Some applications may require that a user program explicitly perform the boxing and unboxing operations that are usually implicit (and invisible) to most programs. The functions that perform these operations are loc and vag respectively. For example, if a user program executes a TENEX JSYS using the ASSEMBLE directive, the value of the ASSEMBLE expression will have to be boxed to be used arithmetically, e.g., (IPLUS X (LOC (ASSEMBLE --))). It must be emphasized that

Arbitrary unboxed numbers should not be passed around as ordinary values because they can cause trouble for the garbage collector.

For example, suppose the value of x were 150000, and you created (vag x), and this just *happened* to be an address on the free storage list! The next garbage collection could be disastrous. For this reason, the function vag must be used with extreme caution when its argument's range is not known.

One place where vag is safe to use is for performing computations on stack positions, which are simply *addresses* of the corresponding positions (cells) on the stack. To treat these addresses as *numbers*, the program must first box them. Conversely, to convert numbers to corresponding stack positions, the program must unbox them. Thus, suppose x were the value of stkarg, i.e., x corresponds to a position on the parameter stack. To obtain the next position on the stack, the program must compute (VAG (ADD1 (LOC X))). Thus if x were #32002,* (LOC X) would be 32002Q,** (ADD1 (LOC X)) 32003Q, and (VAG (ADD1 (LOC X))) #32003.

* A LISP pointer (address) which does not correspond to the address of a list structure, or an atom, or a number, or a string, is printed as #n, n given in octal.

**Q following a number means the numeric quantity is expressed in octal.

Note that rather than starting with a number, and unboxing it to obtain its numeric quantity, here we started with an address, i.e., a 36 bit quantity, and wishing to treat it as a number, boxed it. For example, loc of an atom, e.g., (LOC (QUOTE FOO)), treats the atom as a 36 bit quantity, and makes a number out of it. If the address of the atom FOO were 125000, (LOC (QUOTE FOO)) would be 125000, i.e. the location of FOO. It is for this reason that the box operation is called loc, which is short for location.*

Note that FOO does not print as #364110 (125000 in octal) because the print routine recognizes that it is an atom, and therefore prints it in a special way, i.e. by printing the individual characters that comprise it. Thus (VAG 125000) would print as FOO, and *would be* in fact FOO.

loc[x]	Makes a number out of x, i.e., returns the location of x.
vag[x]	The inverse of loc. x must be a number; the value of vag is the unbox of x.

The compiler eliminates extra vag's and loc's, for example (IPLUS X (LOC (ASSEMBLE --))) will not box the value of the ASSEMBLE, and then unbox it for the addition.

*vag is an abbreviation of value get.

SECTION XIV

INPUT/OUTPUT FUNCTIONS

Contents

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[file]*	Sets file as the primary input file. Its value is the name of the old primary input file.
	input[] is current primary input file, which is not changed.

*The argument name file is used for tutorial purposes only. The arguments to all subrs are U,V, and W as described in arglist, p. 8...

output[file]	Same as input except operates on primary output file.

Any file which is made primary must have been previously opened for input/output, except for the file T, which is always open.

infile[file]	Opens file for input, and sets it as the primary input file.* The value of infile is the previous primary input file. If file is already open, same as input[file]. Generates a FILE WON'T OPEN error if file won't open, e.g., file is already open for *output*.
outfile[file]	Opens file for output, and sets it as the primary output file.* The value of outfile is the previous primary output file. If file is already open, same as output[file]. Generates a FILE WON'T OPEN error if file won't open, e.g., if file is already open for *input*.

For all input/output functions, file follows the TENEX conventions for file names, i.e. file can be prefixed by a directory name enclosed in angle brackets, can contain alt-modes or control-F's, and can include suffixes and/or version numbers. Consistent with TENEX, when a file is opened for input and no version number is given, the highest version number is used. Similarly, when a file is opened for output and no version number is given, a new file is created with a version number one higher than the highest one currently in use with that file name.

*To open file without changing primary input file, perform input[infile[file]]. Similarly for output.

Regardless of the file name given to the LISP function that opened the file, LISP maintains only full TENEX file names* in its internal table of open files and any function whose value is a file name always returns a full file name, e.g. openp[FOO]=FOO.;3. Whenever a file argument is given to an i/o function, LISP first checks to see if the file is in its internal table. If not, LISP executes the appropriate TENEX JSYS to "recognize" the file. If TENEX does not successfully recognize the file, a FILE NOT FOUND error is generated.** If TENEX does recognize the file, it returns to LISP the full file name. Then, LISP can continue with the indicated operation. If the file is being opened, LISP opens the file and stores its (full) name in the file table. If it is being closed, or written to or read from, LISP checks its internal table to make sure the file is open, and then executes the corresponding operation.

Note that each time a full file name is *not* used, LISP must call TENEX to recognize the name. Thus if repeated operations are to be performed, it will be more efficient to obtain the full file name once, e.g. via infilep or outfilep. Also, note that recognition by TENEX is performed on the user's entire directory. Thus, even if only one file is open, say FOO.;1, F$ (F altmode) will not be recognized if the user's directory also contains the file FIE.;1. Similarly, it is possible for a file name that was previously recognized to become ambiguous. For example, a program performs infile[FOO], opening FOO.;1, and reads several expressions from FOO. Then the user types control-C, creates a FOO.;2 and reenters his program. Now a call to read giving it FOO as its file argument will generate a FILE NOT OPEN error, because TENEX will recognize FOO as FOO.;2.

---

*i.e. name, extension, and version, plus directory name if it differs from connected directory.

**except for infilep, outfilep and openp, which in this case return NIL.

infilep[file] Returns full file name of file if recognized by TENEX, NIL otherwise. The full file name will contain a directory field only if the directory differs from the currently attached directory. Recognition is in input context, i.e. if no version number is given, the highest version number is returned.

infilep and outfilep do not open any files, or change the primary files; they are pure predicates.

outfilep[file] Similar to infilep, except recognition is in output context, i.e. if no version number is given, a version number one higher than the highest version number is returned.

closef[file] Closes file. Generates an error if file not open. If file is NIL, it attempts to close the primary input file if other than teletype. Failing that, it attempts to close the primary output file if other than teletype. Failing both, it returns NIL. If it closes any file, it returns the name of that file. If it closes either of the primary files, it resets that primary file to teletype.

closeall[] Closes all open files (except T). Value is a list of the files closed.

openp[file;type] If type=NIL, value is file (full name) if file is open either for reading or for writing. Otherwise value is NIL.

If type is INPUT or OUTPUT, value is file if open for corresponding type, otherwise NIL. If type is BOTH, value is file if open for both input *and* output, (See iofile, p. 14.16) otherwise NIL.

Note: the value of openp is NIL if file is not recognized, i.e. openp does not generate an error.

openp[] is a list of all files open for input or output, excluding T.

Input Functions

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 input file will be used.

Note: in all LISP symbolic files, end of line is indicated by the characters carriage return and line feed in that order. Accordingly, 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 input.*

For all input functions except readc and peekc, when reading from the teletype control-A erases the last character typed in, echoing a \ and the erased character. Control-A will not backup beyond the last carriage return. Typing control-Q causes LISP to print ## and clear the input buffer, i.e. erase the entire line back to the last carriage return.

read[file;flg]

Reads one S-expression from file. Atoms are delimited by parentheses, brackets, double quotes, spaces, and carriage returns. To input an atom which contains one of these syntactic delimiters, precede the delimiter by the escape character %, e.g. AB%(C, is the atom AB(C, %% is the atom %.

Strings are delimited by double quotes. To input a string containing a double quote or a %, precede it by %, e.g. "A B%"C" is the string AB"C. Note that % can always be typed even if next character is not 'special', e.g. %A%B%C is read as ABC.

If an atom is interpretable as a number, read will create a number, e.g. 1E3 reads as a floating point number, 1D3 as a literal atom, 1.Ø as a number,

* Actually, LISP skips the next character after a carriage return without looking at it at all.

1,∅ as a literal atom, etc. Note that an integer can be input in octal by terminating it with a Q, e.g. 17Q and 15 read in as the same integer. The setting of radix, p. 14.18, determines how they are printed.

When reading from the teletype, all input is line-buffered to enable the action of control-Q. Thus no characters are actually seen by the program until a carriage-return is typed. However, for reading by read or uread, when a matching right parenthesis is encountered, the effect is the same as though a carriage return were typed, i.e. the characters are transmitted. To indicate this, LISP also prints a carriage-return line-feed on the teletype.*

read (continued)

flg=T suppresses the carriage-return normally typed by read following a matching right parenthesis. (However, the characters are still given to read - i.e. the user does not have to type the carriage return himself.)

ratom[file]

Reads in one atom from file. Separation of atoms is defined by action of setsepr and setbrk described below. % is also an escape character for ratom, and the remarks concerning control-A, control-Q, and line buffering also apply.

If the characters comprising the atom would normally be interpreted as a number by read, that number is also returned by ratom. Note however that ratom takes no special action for " whether or not it is a break character, i.e. ratom never makes a string.

*Unless control[T] has been performed - pp. 14.19-14.21.

The purpose of ratom, rstring, setbrk, and setsepr is to allow the user to write his own read program without having to resort to reading character by character and then calling pack to make atoms. The function uread (p. 14.10) is available if the user wants to handle input as read does, i.e. same action on parentheses, double quotes, square brackets, dot, spaces, and carriage return, but in addition, to split atoms that contain special characters, as specified by setbrk and setsepr.

rstring[file]	Reads in one string from file, terminated by next break or separator character. Control-A, control-Q, and % have the same effect as with ratom.

Note that the break or separator character that terminates a call to ratom or rstring is not read by that call, but remains in the buffer to become the first character seen by the next reading function that is called.

ratoms[a;file]	Calls ratom repeatedly until the atom a is read. Returns a list of atoms read not including a.
setsepr[lst;flg]	Set separator characters. Value is NIL.
setbrk[lst;flg]	Set break characters. Value is NIL.

For both setsepr and setbrk, lst is a list of character codes. flg determines the action of setsepr/setbrk as follows:

NIL	clear out old tables and reset.
Ø	clear out only those characters in lst - i.e. this provides an unsetsepr and unsetbrk.
1	add characters in lst to corresponding table.

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. For example, if $ was a break character and ! a separator character, the input stream ABC!!DEF$GH!$$ would be read by 6 calls to ratom returning respectively ABC, DEF, $, GH, $, $.

* but have no effect whatsoever on the action of read.

Note that the action of % is not affected by setsepr or setbrk. To defeat the action of % use escape[].

The elements of lst may also be characters e.g. setbrk[(%(%))] has the same effect as setbrk[(40 41)].Note however that the 'characters' 1,2...9,Ø will be interpreted as character *codes* because they are numbers.

getsepr[]	Value is a list of separator character codes.
getbrk[]	Value is a list of break character codes.
escape[flg]	If flg=NIL, makes % act like every other character. Normal setting is escape[T]. The value of escape is the previous setting.
ratest[x]	If x = T, ratest returns T if a separator was encountered immediately prior to the last atom read by ratom, NIL otherwise. If x = NIL, ratest returns T if last atom read by ratom was a break character, NIL otherwise. If x = 1, ratest returns T if last atom read (by read or ratom) contained a % (as an escape character, e.g., %[or %A%B%C), NIL otherwise.

readc[file]

Reads the next character, including %, ", etc. Value is the character. Action of readc is subject to line-buffering, i.e. readc will not return a value until the line has been terminated even if a character has been typed (unless control[T] has been executed, see pp. 14.19-14.21).

peekc[file]

Value is the next character, but does not remove it from the buffer. Not subject to line-buffering, i.e. returns as soon as a character has been typed.

uread[file;flg]

(for user read). Same as read except uses separator and break characters set by setsepr and setbrk. This function is useful for reading in list structure in the normal way, while splitting atoms containing special characters. Thus with space a separator character, and break characters of ( ) . and ' the input stream (IT'S EASY.) is read by uread as the list (IT ' S EASY %.)

Note that ( ) [ ] and " must be included in the break characters if uread is to take special action on them, i.e. assemble lists and make strings.

flg=T suppresses carriage return normally typed following a matching right parentheses. See p. 14.7.

readp[file] — Value is T if there is anything in the input buffer of file, NIL otherwise. (not particularly meaningful for file other than T). Note that because of line buffering, readp may return T even though read may have to wait.

Note: read, ratom, ratoms, peekc, readc, and uread all wait for input if there is none. If reading from a file and an end of file is encountered, they all close the file and generate an error.

readline[]* — reads a line, returning it as a list. If readp[T] is NIL, readline returns NIL. Otherwise it reads, using read, up to the end of the line, as indicated by one of three conditions:

(1) a carriage return, e.g.

A B C↵

and readline returns (A B C)

(2) a list, in which case the list is included in the value of readline, e.g.

A B (C D)

and readline returns (A B (C D))

(3) an unmatched right parentheses or right square bracket, which is not included in the value of readline, e.g.

A B C]

and readline returns (A B C)

*Readline actually has two arguments for use by the system, but the user should consider it as a function of no arguments.

Output Functions

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 output file will be used.

Note: in all LISP symbolic 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.

prin1[x;file]	prints x on file.
prin2[x;file]	prints x on file with %'s and "'s inserted where required for it to read back in properly by read.

Both prin1 and prin2 print lists as well as atoms and strings; neither print a carriage return upon termination; both have value x. prin1 is usually used only for explicitly printing formatting characters, e.g. (PRIN1 (QUOTE %[)) might be used to print a left square bracket (the % would not be printed by prin1). prin2 is used for printing S-expressions which can then be read back into LISP with read i.e. regular LISP formatting characters in atoms will be preceded by %'s, e.g. the atom '()' is printed as %(%) by prin2. If radix=8, prin2 prints a Q after integers but prin1 does not (but both print the integer in octal).

prin3[x;file]	Prints x with %'s and "'s inserted where required for it to read back in properly by uread, i.e. uses separator and break characters specified by setbrk and setsepr to determine when to insert %'s.
print[x;file]	Prints the S-expression x using prin2; followed by a carriage-return linefeed. Its value is x.

For all printing functions, pointers other than lists, strings, atoms, or numbers, are printed as #N, where N is the octal representation of the address of the pointer (regardless of radix). Note that this will not read back in correctly, i.e., it will read in as the atom '#N'.

spaces[n;file]	Prints n spaces; its value is NIL.
terpri[file]	Prints a carriage return; its value is NIL.

Printlevel

The print functions print, prin1, 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 1000. printlevel[] gives current setting.

The variable n controls the number of unpaired left parentheses which will be printed. Below that level, all lists 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, just

```
     &
```

If printlevel is *negative*, the action is similar except that a carriage return is inserted between all occurrences of right paren followed by left paren.

The printlevel setting can be changed dynamically, i.e. while LISP is printing, by typing control-P followed by a number, i.e. a string of digits, followed by a period or exclamation point.* The printlevel will immediately be set to this number.** 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 control-PØ. will cause the list to be terminated.

If a period is used to terminate the printlevel setting, the printlevel will be returned to its previous setting after this printout. If an exclamation point is used, the printlevel is not restored, i.e. the change is permanent (until it is changed again).

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

---

* As soon as control-P is typed, LISP clears and saves the input buffer, clears the output buffer, rings the bell indicating it has seen the control-P, and then waits for input which is terminated by any non-number. The input buffer is then restored and the program continues. If the input was terminated by other than a period or an exclamation point, it is ignored and printing will continue, except that characters cleared from the output buffer will have been lost.

** Another way of "turning off" output is to type control-O, which simply clears the output buffer, thereby effectively skipping the next (up to) 64 characters.

Addressable Files

For most applications, files are read starting at their beginning and proceeding sequentially, i.e. the next character read is the one immediately following the last character read. Similarly, files are written sequentially. A program need not be aware of the fact that there is a file pointer associated with each file that points to the location where the next character is to be read from or written to, and that this file pointer is automatically advanced after each input or output operation. This section describes a function which can be used to *reposition* the file pointer, thereby allowing a program to treat a file as a large block of auxiliary storage which can be accessed randomly.* For example, one application might involve writing an expression at the *beginning* of the file, and then reading an expression from a specified point in its *middle*.**

A file used in this fashion is much like an array in that it has a certain number of addressable locations that characters can be put into or taken from. However, unlike arrays, files can be enlarged. For example, if the file pointer is positioned at the end of a file open for output, and anything is written, the file "grows." It is also possible to position the file pointer *beyond* the end of file and then to write.*** In this case, the file is enlarged, and a "hole" is created, which can later be written

*Random access means that any location is as quickly accessible as any other. For example, an array is randomly accessible, but a list is not, since in order to get to the nth element you have to sequence through the first n-1.

**This particular example requires the file be open for *both* input and output. This can be achieved via the function iofile described below. However, random file input or output can be performed on files that have been opened in the usual way by infile or outfile.

***If the program attempts to *read* beyond the end of file, an error occurs.

into. Note that this enlargement only takes place at the *end* of a file; it is not possible to make more room in the middle of a file. In other words, if expression A begins at position 1000, and expression B at 1100, and the program attempts to overwrite A with expression C, which is 200 characters long, part of B will be clobbered.

iofile[file] Opens file for both input and output. Value is file. Does not change either primary input or primary output. If no version number is given, default is same as for infile, i.e. highest version number.

sfptr[file;address] Sets file pointer for file to address.* Value is old setting. address=-1 corresponds to the end of file. sfptr[file] i.e. address=NIL, returns current value of file pointer without changing it.

filepos[x;file;start;end;skip;tail] Searches file for x a la strpos. (p. 10.7) Search begins at start or current position of file pointer, and goes to end or end of file. Value is *address* of start of match, or NIL if not found. skip can be used to specify a character which matches any character in the file. If tail is T, value if successful is the position of the first character *after* the sequence of characters corresponding to x, not the starting position of the sequence.

* TENEX uses byte addressing; the address of a character (byte) is the number of characters (bytes) that precede it in the file, i.e., Ø is the address of the beginning of the file. However, the user should be careful about computing the space needed for an expression, since end-of-line is represented as two characters in a file, but nchars only counts it as one.

Input/Output Control Functions

clearbuf[file;flg]	Clears the input buffer for file. If file is T and flg is T, contents of LISP's line buffer and the system buffer are saved (internally). When either control-D, control-E, control-H, control-P, or control-S is typed, LISP automatically does a clearbuf[T;T]. (For control-P and control-S, LISP restores the buffer after the interaction. See Appendix 3).
linbuf[flg]	if flg=T, value is LISP's line buffer (as a string) that was saved at last clearbuf[T;T]. If flg=NIL, clears this internal buffer.
sysbuf[flg]	a la linbuf.

The internal buffers associated with linbuf and sysbuf are not changed by a clearbuf[T;T] if both LISP's line buffer and the system buffer are empty.

bklinbuf[x]	x is a string. bklinbuf sets LISP's line buffer to x. If greater than 160 characters, first 160 taken.
bksysbuf[x]	x is a string. bksysbuf sets system buffer to x. If greater than 63 characters, first 63 are taken. The effect is the same as though the user typed x.

Note that bklinbuf, bksysbuf, linbuf, and sysbuf provide a way of 'undoing' a clearbuf. Thus if the user wants to "peek" at various characters in the buffer, he could perform clearbuf[T;T], examine the buffers via linbuf and sysbuf, and then put them back.

radix[n]

Resets output radix* to $|n|$ with sign indicator the sign of n. For example, -9 will print as shown with the following radices

radix	printing
10	-9
-10	68719476727 i.e. $(2^{36}-9)$
8	-11Q
-8	777777777767Q

Value of <u>radix</u> is last setting. radix[] gives current setting without changing it. Initial setting is 10.

linelength[n]

Sets the length of the print line for all files. Value is the former setting of the line length. Whenever printing an atom would go *beyond* the length of the line, a carriage return is automatically inserted first.

linelength[] gives current setting. Initial setting is 72.

position[file]

Gives the column number the next character will be read from or printed to, e.g. after a carriage return, position=Ø. Note that position[file] is *not* the same as sfptr[file] which gives the position in the *file*, not on the line.

* Currently there is no input radix.

Control[] (Normal State)

In LISP's normal state, characters typed on the teletype (this section does not apply in any way to input from a file) are transferred to a line-buffer. Characters are transmitted from the line buffer to whatever input function initiated the request (i.e., read, uread, ratom, or readc)* *only* when a carriage return is typed. Until this time, the user can delete characters one at a time from the input buffer by typing control-A. The characters are echoed preceded by a \. Or, the user can delete the entire line buffer back to the last carriage return by typing control-Q, in which case LISP echoes ##. (If no characters are in the buffer and either control-A or control-Q is typed, LISP echoes ##.)

Note that this line editing is *not* performed by read or ratom but by LISP, i.e. it does not matter (nor is it necessarily known) which function will ultimately process the characters, only that they are still in the LISP input buffer. Note also that it is the function that is *currently* requesting input that determines whether parentheses counting is observed, e.g. if the user executes (PROGN (RATOM) (READ)) and types in A (B C D) he will have to type in the carriage return following the right parenthesis before any action is taken, whereas if he types (PROGN (READ) (READ)) he would not. However, once a carriage return has been typed, the entire line is 'available' even if not all of it is processed by the function initiating the request

---

* peekc is an exception, it returns the character immediately.

** As mentioned earlier, for calls from read or uread, the characters are also transmitted whenever the parentheses count reaches 0. In this case, if the second argument to read or uread is NIL, LISP also outputs a carriage-return line-feed.

for input, i.e. if any characters are 'left over' they will be returned immediately on the next request for input. For example, (PROGN (RATOM) (READC)) followed by A B carriage return will perform both operations.

Control[T]

The function control is available to defeat this line buffering. After control[T], characters are returned to the calling function without line-buffering as described below. The function that initiates the request for input determines how the line is treated:

1. read/uread

if the expression being typed is a list, the effect is the same as though control were NIL, i.e. line buffering until carriage return or matching parentheses. If the expression being typed is not a list, it is returned as soon as a break or separator character is encountered,* e.g. (READ) followed by ABC space will immediately return ABC. Control-A and control-Q editing are available on those characters still in the buffer. Thus, if a program is performing several reads under control[T] and the user types NOW IS THE TIME followed by control-Q, he will delete only TIME since the rest of the line has already been transmitted to read and processed.

* An exception to the above occurs when the break or separator character is a (,", or [,since returning at this point would leave the line buffer in a "funny" state. Thus if control is T and (READ) is followed by 'ABC(', the ABC will not be read until a carriage return or matching parentheses is encountered. In this case the user could control-Q the entire line, since all of the characters are still in the buffer.

2. ratom

characters are returned as soon as a break or separator character is encountered. Before then, control-A and control-Q may be used as with **read**, e.g. (RATOM) followed by ABCcontrol-Aspace will return AB. (RATOM) followed by (control-A will return (and type ## indicating that control-A was attempted with nothing in the buffer, since the (is a break character and would therefore already have been read.

3. readc/peekc

the character is returned immediately; no line editing is possible. In particular, (READC) followed by control-A will read the control-A, (READC) followed by & will read the &.

control[u]	u=T	eliminates LISP's normal line-buffering.
	u=NIL	restores line buffering (normal).
	u=0	eliminates echo of character being deleted by control-A.
	u=1	restores echo (normal).

Special Functions

sysout[file]	Saves the user's private memory on file. Also saves the stacks, so that if a program performs a sysout, the subsequent sysin will continue from that point, e.g. `(PROGN (SYSOUT (QUOTE FOO)) (PRINT (QUOTE HELLO)))` will cause HELLO to be printed after (SYSIN (QUOTE FOO)) The value of sysout is file (full name). A value of NIL indicates the sysout was unsuccessful, i.e., either disk or computer error, or user's directory was full.

Sysout does not save the state of any open files.

Whenever the LISP system is reassembled and/or reloaded, old sysout files are not compatible.

sysin[file]	restores the state of LISP from a sysout file. Value is T. If sysin returns NIL, there was a problem in reading the file. If the file was not compatible (see sysout above), generates an error.

Since sysin continues immediately where sysout left off, the only way for a program to determine whether it is just coming back from a sysin or from a sysout is to test the value of sysout, e.g.

(COND ((EQ (SYSOUT (QUOTE FOO)) T) (PRINT (QUOTE HELLO)))) will cause HELLO to be printed following the sysin but not when the sysout was performed.

Symbolic File Input

load[file;dfnflg;printflg]

Reads successive S-expressions from file and evaluates each as it is read, until it reads either NIL, or the single atom STOP. Value is file (full name).

If printflg=T, load prints the value of each S-expression; otherwise it does not. dfnflg=NIL or T affects the operation of defineq expressions as described on p. 8.7. However, if dfnflg=PROP, defineq is *not* called. Instead, the function definitions are stored on the property lists under the property EXPR.

readfile[file]

Reads successive S-expressions from file using read until the single atom STOP is read, or an end of file encountered. Value is a list of these S-expressions.

Symbolic File Output

writefile[x;file;dateflg]

Writes successive S-expressions from x onto file. If x is atomic, its value is used. If file is not open, it is opened. If the first expression on x is the type produced by printdate, or if dateflg is T, the current date is written. If file is a list, car[file] is used and the file is left opened. Otherwise, when x is finished, a STOP is printed on file and it is closed. Value is file.

pp[x]

nlambda, nospread function that performs output[T] and then calls prettyprint:
PP FOO is equivalent to PRETTYPRINT((FOO))
PP(FOO FIE) or (PP FOO FIE) is equivalent to PRETTYPRINT((FOO FIE))
Primary output file is restored after printing.

prettyprint[x]*

x is a list of functions (if atomic, its value is used). The definitions of the functions are printed in a pretty format on the primary output file.

Example:

```
(FACTORIAL
  [LAMBDA (N)
    (COND
      ((ZEROP N)
        1)
      (T (ITIMES 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, but does not change the current state of the function.

Comment Feature

A facility for annotating 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!)
         (ITIMES N (FACTORIAL (SUB1 N])
```

*prettyprint has a second argument that is T when called from prettydef. In this case, whenever prettyprint starts a new function, it prints (on the teletype) the name of that function if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

These comments actually form a part of the function definition. Accordingly, * is defined as an NLAMBDA NOSPREAD function that returns its argument, i.e. it is equivalent to quote. When running an interpreted function, * is entered the same as any other LISP function. Therefore, comments should only be placed where they will not harm the computation i.e. where a quoted expression could be placed. For example, writing
(ITIMES N (FACTORIAL (SUB1 N)) (* RECURSIVE DEFINITION)) in the above function would cause an error when ITIMES 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. For more options, see end of this section.

Comments are designed mainly for documenting *listings*. Thus when prettyprinting to the teletype, comments are suppressed and printed as the atom **COMMENT**.

prettydef

prettydef[prettyfns;prettyfile;prettycoms] Used to make symbolic files that are suitable for loading which contain function definitions, variable settings, property lists, et al, in a prettyprint format.

The arguments are interpreted as follows:

prettyfns (first argument)	Is a list of function names. The functions on the list are prettyprinted surrounded by a (DEFINEQ ...) so that they can be loaded with load. If prettyfns is atomic, its top level value is used as the list of function names, and an rpaqq* will also be written which will set that atom to the list of functions when the file is loaded. A print expression will also be written which informs the user of the named atom or list of functions when the file is subsequently loaded.
prettyfile (second argument)	is the name of the file on which the output is to be written. The following options exist: prettyfile=NIL The primary output file is used. prettyfile atomic The file is opened if not already open, and becomes the primary output file. File is closed at end of prettydef and primary output file restored.

* rpaqq is like setqq except it sets the top level value. Its name comes from rplaca quote quote, since it is an NLAMBDA version of rplaca with both arguments considered as quoted.

prettyfile a list
Car of the list is assumed to be the file name and is opened if not already open. The file is left open at end of prettydef.

prettycoms (third argument)	Is a list of commands interpreted as described below. If prettycoms is atomic, its top level value is used and an rpaqq is written which will set that atom to the list of commands when the file is loaded. A print is written which informs the user of the named atom or list of commands when the file is subsequently loaded, exactly as with prettyfns.

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 atomic, an rpaqq is written which will restore the top level value of this atom when the file is loaded.

2. (PROP propname $atom_1$ ... $atom_n$)
   an appropriate deflist will be written which will restore the value of propname for each $atom_i$ when the file is loaded. If

propname=ALL, the values of all user properties (on the property list of each $atom_i$) are saved.* If propname is a list, deflist's will be written for each property on that list.

3. (ARRAY $atom_1$ ... $atom_n$), each atom following ARRAY should have an array as its value. An appropriate expression will be written which will set the atom to an array of exactly the same size, type, and contents upon loading.

4. (P ...), each S-expression following P will be printed on the output file, and consequently evaluated when the file is loaded.

5. (E ...), each form following E will be evaluated at output time, i.e., when prettydef reaches this command.

6. (FNS fn_1 ...fn_m),, a defineq is written with the definitions of fn_1 ... fn_m exactly as though (fn_1 ... fn_m) 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, he would then write the definitions using the FNS command instead of the first argument to prettydef.

7. (VARS var_1 ... var_n), for each var_i, an expression will be written which will set its top level value when the file is loaded. If var_i is atomic, var_i will be set to the top-level value it had at the time the file was prettydefed, i.e. (RPAQQ var_i top-level-value) is written. If var_i is non-atomic, it is interpreted as (var form).
e.g. (FOO (APPEND FIE FUM)) or (FOO (QUOTE (FOO1 FOO2 FOO3))).
In this case the expression (RPAQ var form)** is written.

*sysprops is a list of properties used by system functions. Only properties *not* on that list are dumped when the ALL option is used.

**rpaq is to rpaqq as setq is to setqq, i.e. the first argument is considered quoted, the second is evaluated. Note that evaluation takes place at *load* time.

8. (ADVISE fn_1 ... fn_m), for each fn_i, an appropriate expression will be written which will reinstate the function to its advised state when the file is loaded.

9. (ADVICE fn_1 ... fn_m), for each fn_i, will write a deflist which will put the advice back on the property list of the function. The user can then use readvise to reactivate the advice. See Chapter 19.

10. (BLOCKS $block_1$ ... $block_n$) for each $block_i$, a declare expression will be written which the block compile functions interpret as block declarations. See Chapter 18.

11. (COMS com_1 ... com_n), each of the commands com_1 ... com_n will be interpreted as one of the eleven command types.

In each of the eleven commands described above, if the atom * follows the command type, the form following the *, i.e., caddr of the command, is evaluated and its value used in executing the command, e.g., (FNS * (APPEND FNS1 FNS2)).† Note that (COMS * form) provides a way of *computing* what should be done by prettydef.

† Except for the PROP command, in which case the * must follow the property name, e.g., (PROP MACRO * FOOMACROS).

Example:

```
SET(FOOFNS (FOO1 FOO2 FOO3))
SET(FOOVARS(FIE (PROP MACRO FOO1 FOO2) (P (MOVD (QUOTE FOO1)
  (QUOTE  FIE1]
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) T)
4. (RPAQQ FOOFNS (FOO1 FOO2 FOO3))
5. (PRINT (QUOTE FOOVARS) T)
6. (RPAQQ FOOVARS (FIE ...)
7. (RPAQQ FIE value of fie)
8. (DEFLIST (QUOTE((FOO1 propvalue) (FOO2 propvalue))) (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*. *minspaces* indicates the minimum number of spaces to be printed by *tab*, i.e., it is intended to be a small number (if NIL, 1 is used). Thus, if position + minspaces is greater than *pos*, *tab* does a *terpri* and then spaces[pos].
endfile[file]	Prints STOP on *file* and closes it.
printdef[e;left]	prints the expression *e* on the primary output file in a pretty format, i.e., *prettyprint* is essentially printdef[getd[fn]]. *left* is the left-hand margin (linelength determines the right hand margin). 2 is used if *left*=NIL.

Special Prettyprint Controls

With the exception of prettyflg, all variables described below, i.e., #rpars, firstcol, et al, are globalvars, see p. 18.6. Therefore, if they are to be changed, they must be *reset*, not rebound.

#rpars	controls the number of right parentheses necessary for square bracketing to occur. If #rpars=NIL, no brackets are used. #rpars is initialized to 4.
linelength[n]	determines the position of the right margin for prettyprint.
firstcol	is the starting column for comments. Initial setting is 48. Comments run between firstcol and linelength. If a word in a comment ends with a '.' and is not on the list abbrevlst, and the position is greater than halfway between firstcol and linelength, the next word in the comment begins on a new line. Also, if a list is encountered in a comment, and the position is greater than halfway, a carriage return is printed.
prettylcom	If a comment is bigger (using count) than prettylcom in size, it is printed starting at column 10, instead of firstcol. prettylcom is initialized to 14 (arrived at empirically).

widepaper[flg]	widepaper[T] sets linelength to 120, *firstcol* to 80 and *prettylcom* to 28. This is a useful setting for pretty-printing files to be listed on wide paper. widepaper[] restores these parameters to their initial values.
commentflg	If *car* of an expression is *eq* to *commentflg*, the expression is treated as a comment. *commentflg* is initialized to *.
prettyflg	If *prettyflg* is NIL, printdef uses *prin2* instead of prettyprinting. This is useful for producing a fast symbolic dump (e.g. when TENEX is very slow.) Initial setting is T.
prettymacros	Is an assoc-type list for defining substitution macros for *prettydef*. If (FOO (X Y) . coms) appears on *prettymacros*, then (FOO A B) appearing in the third argument to *prettydef* will cause A to be substituted for X and B for Y throughout *coms* (i.e., *cddr* of the macro), and then *coms* treated as a list of commands for *prettydef*.

(* E x)

A comment of this form causes x to be evaluated at prettyprint time, e.g., (* E (RADIX 8)) as a comment in a function containing octal numbers can be used to change the radix to produce more readable printout. The comment, of course, is also printed.

Lower Casing comments

%%

If the second atom in a comment is %%, the text of the comment is converted to lower case so that it looks like English instead of LISP (see next page).

The output on the next page illustrates the result of a lower casing operation. Before this function was prettydefed, all comments consisted of upper case atoms, e.g., the first comment was (* %% INTERPRETS A SINGLE COMMAND). Note that comments are converted *only* when they are actually written to a file by prettydef, and that only the line printer can print lower case characters, i.e., lower case characters are printed as upper case on teletypes.

The algorithm for conversion to lower case is the following: If the first character in an atom is ↑, do not change the atom (but remove the ↑). If the first character is %, convert the atom to lower case.* If the atom** is a LISP word,*** do not change it. Otherwise, convert the atom to lower case.

* User must type %% as % is the escape character.

** minus any trailing punctuation marks.

*** i.e., is a bound or free variable for the function containing the comment, or has a top level value, or is a defined function, or has a non-NIL property list.

```
(BREAKCOM
  (LAMBDA (BRKCOM BRKFLG)                             (* Interprets a single
                                                      command.)
    (PROG (BRKZ)
      TOP (SELECTQ
            BRKCOM
            [↑ (RETEVAL (QUOTE BREAK1)
                        (QUOTE (ERROR!]
            (GO                                       (* Evaluate BRKEXP
                                                      unless already
                                                      evaluated, print value,
                                                      and exit.)
                (BREAKCOM1 BRKEXP BRKCOM NIL BRKVALUE)
                (BREAKEXIT))
            (OK                                       (* Evaluate BRKEXP,
                                                      unless already
                                                      evaluated, do NOT print
                                                      value, and exit.)
                (BREAKCOM1 BRKEXP BRKCOM BRKVALUE BRKVALUE)
                (BREAKEXIT T))
            (←WGO                                     (* Same as GO except
                                                      never saves evaluation
                                                      on history.)
                (BREAKCOM1 BRKEXP BRKCOM T BRKVALUE)
                (BREAKEXIT))
            (RETURN

         (* User will type in expression to be evaluated and
         returned as value of BREAK.
         Otherwise same as GO.)


                  (BREAKCOM1 [SETQ BRKZ (COND
                                          (BRKCOMS (CAR BRKCOMS))
                                          (T (LISPXREAD T]
                             (QUOTE RETURN)
                             NIL NIL (LIST (QUOTE RETURN)
                                           BRKZ))
                  (BREAKEXIT))
            (EVAL                                     (* Evaluate BRKEXP but
                                                      do not exit from BREAK.)
                (BREAKCOM1 BRKEXP BRKCOM)
                (COND
                  (BRKFLG (BREAK2)
                          (PRIN1 BRKFN T)
                          (PRIN1 (QUOTE " EVALUATED
")
                                 T)))
```

Conversion only affects the upper case alphabet, i.e., atoms already converted to lower case are not changed if the comment is converted again. When converting, the first character in the comment, and the first character following each period, are left capitalized. After conversion, the comment is physically modified to be the lower case text minus the %% flag, so that conversion is thus only performed once (unless the user edits the comment inserting additional upper case text and another %% flag).

lcaselst	Words on lcaselst will always be converted to lower case. lcaselst is initialized to contain words which are LISP functions but also appear frequently in LISP comments as English words. e.g. AND, EVERY, GET, GO, LAST, LENGTH, LIST, etc. Thus in the example on the previous page, not was written as ↑NOT, and go as ↑GO in order that they might be left in upper case.
ucaselst	words on ucaselst (that do not appear on lcaselst) will be left in upper case. ucaselst is initialized to NIL.
abbrevlst	abbrevlst is used to distinguish between abbreviations and words that ends in periods. Normally, words that end in periods and occur more than halfway to the right margin cause carriage returns. Furthermore, during conversion to lowercase, words ending in periods, except for those on abbrevlst, cause the first character in the *next* word to be capitalized. abbrevlst is initialized to the upper and lower case forms of ETC. I.E. and E.G.
l-case[x;flg]	value is lower case version of x. If flg is T, the first letter is capitalized, e.g. l-case[TEST;T]=Test, l-case[TEST]=test
u-case[x]	Similar to l-case

Special Edit Commands for Editing Lower Case Comments

RAISE	is an edit macro that is defined as UP followed by (I 1 (U-CASE (## 1))), i.e. it raises to upper-case the current expression in the editor, or if a tail, the first element of the current expression.
LOWER	similar to RAISE
(RAISE x)	equivalent to (R lower-case-of-x X) i.e. changes every lower-case x to uppercase.
(LOWER x)	similar to (RAISE X)
CAP	First does a RAISE and then lowers all but the first character, i.e., the first character is left capitalized. Note that RAISE, LOWER, and CAP are always NOPs if the atom is already in that state.
(%%F x)	Is an edit macro for doing a lower-case find, e.g. (%%F FOO) will find the lower case version of FOO. Equivalent to (I F (L-CASE X) (QUOTE N))
(%%F x T)	Finds the lower-case capitalized version of FOO, e.g. (%%F FINDS T) Searches for 'Finds'.

%%	Is an edit command that first converts CDDR of the command as though it were a comment, and then executes the command, e.g. (%% N OTHERWISE, RETURNS NIL.) will attach the lowercase versions of the indicated words at the end of the current expression.

File Package

This section describes a set of functions and conventions for facilitating the bookkeeping involved with working in a large system consisting of many symbolic files and their compiled counterparts, i.e. it keeps track of which files have been in some way modified and need to be dumped, which files have been dumped, but still need to be listed and/or recompiled. The functions described below comprise a coherent package for eliminating this burden from the user. They require that for each file the first argument to prettydef, (if any), be an atom of the form fileFNS, and the third argument, (if any), be fileVARS where file is the name of the file, e.g. prettydef[FOOFNS; FOO; FOOVARS].*

The functions load, editf, editv, tcompl, recompile, bcompl, and brecompile interact with the functions and global variables in the file package as follows. Whenever load is called, its argument is added to the list filelst, and the property FILE, value (fileFNS fileVARS), is added to the property list of the file name.** This property value is used to determine whether or not the file has been modified since the last time it was loaded or dumped. Whenever the user calls editf, and exits via OK, filelst is searched to find the file which defined this function, i.e. the file for which the function was either a member of fileFNS, or appeared in a FNS command on fileVARS. When (if) the corresponding file is found, the name of the function is added, using nconc, to the value of the property FILE for that file. Thus if the user loads the file FOO containing definitions for FOO1, FOO2, and FOO3, and then edits FOO2, getp[FOO;FILE] will be (FOOFNS FOOVARS FOO2) following the edit. A similar update takes place for calls to editv.

---

*file can contain a suffix and/or version number, e.g. prettydef[FOOFNS; FOO.TEM;3 FOOVARS] is acceptable. The essential point is that the FNS and VARS be computable from the name of the file.

**The name added to filelst has the version number removed, if any, and is added at the front of filelst (if already on filelst, it is moved to its front.) fileFNS and fileVARS are constructed using only the name field, ie., if the user performs load[FOO.TEM;2], FOO.TEM is added to filelst, and (FOOFNS FOOVARS) put on the property list of FOO.TEM.

Whenever the user dumps a file using makefile (described below), the file is added to filelst (if not already there) and its FILE property is reinitialized to (fileFNS fileVARS), indicating that the file is up to date. In addition, the file is added to the list notlistedfiles and notcompiledfiles. Whenever the user lists a file using listfiles, it is removed from notlistedfiles. Similarly, whenever a file is compiled by tcompl, recompile, bcompl, or brecompile, the file is removed from notcompiledfiles. Thus at each point, the state of all files can be determined. This information is available to the user via the function files?. Similarly, the user can see whether and how each particular file has been modified, dump all files that have been modified, list all files that have been dumped but not listed, recompile all files that have been dumped but not recompiled, or any combination of any or all of the above by using one of the functions described below.

makefile[file;options] — adds file to filelst if not already there. Calls prettydef[fileFNS;file;fileVARS].* adds file to notlistedfiles, notcompiledfiles. options is a list of options interpreted as follows (if atomic and non-nil, it is treated as (options)):

FAST	perform prettydef with prettyflg=NIL
RC	call recompile or brecompile after prettydef. Choice depends on whether fileBLOCKS is NOBIND.
C	calls tcompl or bcompl after prettydef. Choice depends on whether fileBLOCKS is NOBIND.
LIST	calls listfiles on file.

*fileFNS and fileVARS are constructed from the name field only, e.g. makefile[FOO.TEM] will work.

For the three compile options, ST is used for the answer to the compiler's question LISTING?, unless F is given as the *next* option, e.g. makefile[FOO; (TC F LIST)] will dump FOO, then TCOMPL it without redefining any functions, and finally list the file.

makefiles[options]

For each file on *filelst* that has been changed, performs makefile[file;options], e.g. makefiles[LIST] will make and list all files. Value is a list of all files that are made.

listfiles[files]

nlambda, nospread function. Uses *bksysbuf* to load system buffer appropriately to list each file on *files*, (if NIL, *notlistedfiles* is used) and then to CONTINUE, then does a *logout*. TENEX then reads from the system buffer, lists the files, and CONTINUES the program.

Each file listed is removed from *notlistedfiles* if the listing is completed, e.g. if LPT NOT MOUNTED is typed and user then does a CONTINUE, *listfiles* will not remove the files from *notlistedfiles*. Similarly if user control-C's to stop the listing and CONTINUES.

files?[]	Prints on teletype the names of those files that have been modified but not dumped, dumped but not listed, dumped but not compiled.
cleanup[]	Performs makefiles[], followed by listfiles[], followed by either *recompile* or *brecompile* for all files on *notcompiledfiles*.

SECTION XV

DEBUGGING - THE BREAK PACKAGE

Contents

Debugging Facilities

Debugging a collection of LISP functions involves isolating problems within particular functions and/or determining when and where incorrect data are 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, break1 described below.

Break modifies the definition of its argument, a function 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 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.

```
←PP FACTORIAL

(FACTORIAL
  [LAMBDA (N)
    (COND
      ((ZEROP N)
        L)
      (T (ITIMES N (FACTORIAL (SUB1 N])
FACTORIAL
←TRACE(FACTORIAL)
(FACTORIAL)
←FACTORIAL(4)

FACTORIAL:
N = 4


   FACTORIAL:
   N = 3


      FACTORIAL:
      N = 2


         FACTORIAL:
         N = 1


            FACTORIAL:
            N = 0

U.B.A.
L
(FACTORIAL BROKEN)
:N
0
:RETURN 1
            FACTORIAL = 1
         FACTORIAL = 1
      FACTORIAL = 2
   FACTORIAL = 6
FACTORIAL = 24
24
←
```

```
←PP FACTORIAL

(FACTORIAL
  [LAMBDA (N)
    (PROG ((M 1))
      LOOP(COND
            ((ZEROP N)
              (RETURN M)))
          (SETQ M (ITIMES M N))
          (SETQ N (SUB1 N))
          (GO LOOP])
FACTORIAL
←BREAKIN(FACTORIAL (AFTER LOOP) (ILESSP N 2]
SEARCHING...
FACTORIAL
←FACTORIAL(5)

((FACTORIAL) BROKEN)
:NN
U.B.A.
NN
(FACTORIAL BROKEN AFTER LOOP)
:N
1
:M
120
:OK
(FACTORIAL)

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

Break1

The basic function of the break package is break1. Whenever LISP types a message of the form (- BROKEN) followed by ':' the user is then 'talking to' break1, and we say he is 'in a break.' break1 allows the user to interrogate the state of the world and affect the course of the computation. It uses the prompt character ':' to indicate it is ready to accept input(s) for evaluation, in the same way as evalqt uses '←'. The user may type in an expression for evluation as with evalqt, and the value will be printed out, followed by another :. Or the user can type in one of the commands specifically recognized by break1 described below.

Since break1 puts all of the power of LISP at the user's command, he can do anything he can do at evalqt. For example, he can insert new breaks on subordinate functions simply by typing:

```
(BREAK fn1 fn2 ...)
```

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

```
(UNBREAK fn3 fn4 ...)
```

He can edit functions, including the one currently broken:

```
EDITF(fn)
```

For example, the user might evaluate an expression, see that the value was incorrect, call the editor, change the function, and evaluate the expression again, all without leaving the break.

Similarly, the user can prettyprint functions, define new functions or redefine old ones, load a file, compile functions, time a computation, etc. In short, anything that he can do at the top level 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 or reteval.

It is important to emphasize that once a break occurs, the user is in complete control of the flow of the computation, and the computation will not proceed without specific instruction from him. If the user types in an expression whose evaluation causes an error, the break is maintained. Similarly if the user aborts a computation* initiated from within the break, the break is maintained. Only if the user gives one of the commands that exits from the break, or evaluates a form which does a _retfrom_ or _reteval_ back out of _breakl_, will the computation continue.**

Note that _breakl_ is just another LISP function, not a special system feature like the interpreter or the garbage collector. It has arguments which are explained later, and returns a value, the same as _cons_ or _cond_ or _prog_ or any other function. The value returned by _breakl_ is called 'the value of the break.' The user can specify this value explicitly by using the RETURN command described below. But in most cases, the value of a break is given implicitly, via a GO or OK command, and is the result of evaluating 'the break expression,' _brkexp_, which is one of the arguments to _breakl_.

The break expression is an expression equivalent to the computation that would have taken place had no break occurred. For example, if the user breaks on the function FOO, the break expression is the body of the definition of FOO. When the user types OK or GO, the body of FOO is evaluated, and its value returned as the value of the break, i.e. to whatever function called FOO. The effect is the same as though no break had occurred. In other words, one can think of _breakl_ as a fancy _eval_, which permits interaction before and after evaluation. The break expression then corresponds to the argument to _eval_.

*By typing control-E, see Section 16.

**Except that breakl does not 'turn off' control-D, i.e. a control-D will force an immediate return back to the top level.

Break Commands

GO	Releases the break and allows the computation to proceed. break 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 equivalent to the body of the definition of the broken function. For breakin, using BEFORE or AFTER, brkexp is NIL. For breakin AROUND, brkexp is the indicated expression. See breakin, p. 15.21.
OK	Same as GO except the value of brkexp is not printed.
EVAL	Same as GO or OK except that the break is maintained after the evaluation. The user can then interrogate the value of the break 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. EVAL is a useful command 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 form
or
RETURN fn[args]
	The value of the indicated computation is returned as the value of the break. For example, one might use the EVAL command and follow this with RETURN (REVERSE !VALUE).

↑	Calls error! 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 commands, maintain the break.

!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 equivalent to !EVAL followed by OK.

!GO

Function is first unbroken, evaluated, rebroken, and exited with value typed, i.e. !EVAL followed by GO.

UB

unbreaks brkfn, e.g.

```
(FOO BROKEN)
:UB
FOO
```

and FOO is now unbroken

@

resets the variable lastpos, which establishes a context for the commands ?=, ARGS, BT, BTV, BTV*, and EDIT, and IN? described below. lastpos is the position of a function call on the push-down stack. It is initialized to the function just before the call to break1, i.e. stknth[-1;BREAK1]

@ treats the rest of the teletype line as its argument(s). It first resets lastpos to stknth[-1;BREAK1] and then for each atom on the line, @ searches backward, for a call to that atom. The following atoms are treated specially:

@ do not reset lastpos to stknth[-1;BREAK1] but leave it as it was, and continue searching from that point.

numbers	if negative, move <u>lastpos</u> back that number of calls, if positive, forward, i.e. reset <u>lastpos</u> to stknth[n;lastpos]
←	search <u>forward</u> for next atom
/	the next atom is a number and can be used to specify more than one call e.g. @ FOO / 3 is equivalent to @ FOO FOO FOO

Example:

if the push-down stack looks like

```
BREAK1    (13)
FOO       (12)
SETQ      (11)
COND      (10)
PROG       (9)
FIE        (8)
COND       (7)
FIE        (6)
COND       (5)
FIE        (4)
COND       (3)
PROG       (2)
FUM        (1)
```

then @ FIE COND will set <u>lastpos</u> to the position corresponding to (7); @ @ COND will then set <u>lastpos</u> to (5). @ FUM ← FIE will stop at (4). @ FIE / 3 -1 will stop at (3).

If @ cannot successfully complete a search, it types (fn NOT FOUND) where fn is the name of the function for which it was searching.

When @ finishes, it types the name of the function at <u>lastpos</u>, i.e. stkname[lastpos]

@ can be used on <u>brkcoms</u>. In this case, the <u>next</u> command on <u>brkcoms</u> is treated the <u>same</u> as the rest of the teletype line.

?=

This is a multi-purpose command. Its most common use is to interrogate the value(s) of the arguments of the broken function, e.g. if FOO has three arguments (X Y Z), then typing ?= to a break on FOO, will produce

```
:?=
X =    value of X
Y =    value of Y
Z =    value of Z
```

?= operates on the rest of the teletype line as its arguments. If the line is empty, as in the above case, it prints all of the arguments. If the user types ?= X (CAR Y), he will see the value of X, and the value of (CAR Y). The difference between using ?= and typing X and (CAR Y) directly to <u>breakl</u> is that ?= evaluates its inputs as of <u>lastpos</u>, i.e. it uses <u>stkeval</u>. This provides a way of examining variables or performing computations *as of a particular point on the stack*. For example, @ FOO / 2 followed by ?= X will allow the user to examine the value of X in the previous call to FOO, etc.

?= also recognizes numbers as referring to the correspondingly numbered argument, i.e. it uses <u>stkarg</u> in this case. Thus

```
:@ FIE
:?= 2
```

will print the name and value of the second argument of FIE.

?= can also be used on <u>brkcoms</u>, in which case the next command on <u>brkcoms</u> is treated as the rest of the teletype line. For example, if <u>brkcoms</u> is (EVAL ?= (X Y) GO), <u>brkexp</u> will be evaluated, the values of X and Y printed, and then the function exited with its value being printed.

BT	Prints a backtrace of *function names only* starting at lastpos. (See @.) The several nested calls in system packages such as break, edit, and the top level executive appear as the single entries **BREAK**, **EDITOR**, and **TOP** respectively.
BTV	Prints a backtrace of function names *with* variables beginning at lastpos.
BTV*	Same as BTV except also prints arguments of internal calls to eval. (See section 12)
BTV!	Same as BTV except prints *everything* on stack. (See section 12).

BT, BTV, BTV*, and BTV! all permit an optional functional argument which is a predicate that chooses functions to be *skipped* on the backtrace, e.g., BT SUBRP will skip all SUBRS, BTV (LAMBDA (X) (NOT (MEMB X FOOFNS))) will skip all but those functions on FOOFNS. If used as a brkcom, the functional argument is no longer optional, i.e., the next brkcom must either be the functional argument, or NIL if no functional argument is to be applied.

For BT, BTV, BTV*, and BTV!, if Control-P is used to change a printlevel during the backtrace, the printlevel will be restored after the backtrace is completed.

ARGS	Prints the names of the variables bound at lastpos, i.e. variables[lastpos] (p. 12.10). For most cases, these are the arguments to the function entered at that position, i.e. arglist[stkname[lastpos]].

The following two commands are for use only with unbound atoms or undefined function breaks (see Section 16.)

= form or = fn[args]	*only* for the break following an unbound atom error. Sets the atom to the value of the form, or function and arguments, exits from the break returning that value, and continues the computation, e.g. U.B.A. (FOO BROKEN) := (COPY FIE) sets FOO and goes on.
-> expr	for use either with unbound atom error, or undefined function error. Replaces the expression containing the error with expr* (not the value of expr) e.g., U.D.F. (FOO1 BROKEN) :->FOO changes the FOO1 to FOO and continues the computation. expr need not be atomic, e.g. U.B.A. (FOO BROKEN) :-> (QUOTE FOO) For U.D.F. breaks, the user can specify a function and its first argument, e.g. U.D.F. (MEMBERX BROKEN) :->MEMBER X Note that in the case of a U.D.F. error occurring immediately following a call to apply, e.g. (APPLY X Y) where value of x is FOO and FOO is undefined, or a U.B.A. error immediately following a call to eval, e.g. (EVAL X), value of x is FOO and FOO is unbound, there is no expression containing the offending atom. In this case, ? is printed and no action taken.

*-> does not change just brkexp; it changes the function or expression containing the erroneous form. In other words, the user does not have to perform any additional editing.

EDIT designed for use in conjunction with breaks caused by errors. Facilitates editing the expression causing the break:

```
NON-NUMERIC ARG
NIL
(IPLUS BROKEN)
:EDIT
IN FOO...
(IPLUS X Z)
EDIT
*(3 Y)
*OK
FOO
:
```

and user can continue by typing OK, EVAL, etc.

This command is very simple conceptually, but complicated in its implementation by all of the exceptional cases involving interactions with compiled functions, breaks on user functions, error breaks, breaks within breaks, et al. Therefore, we shall give the following simplified explanation which will account for 90% of the situations arising in actual usage. For those others, EDIT will print an appropriate failure message and return to the break.

EDIT begins by searching up the stack beginning at lastpos (set by @ command, initially position of break) looking for a form, i.e. an internal call to eval. Then EDIT continues from that point looking for a call to an interpreted function, or to eval. It then calls the editor on either the EXPR or the argument to eval in such a way as to look for an expression eq to the form that it first found. It then prints the form, and permits interactive editing to begin. Note that the user can then type successive 0's to the editor to see the chain of superforms for this computation.

If the user exits from the edit with an OK, the break expression is reset, if possible, so that the user can continue with the computation by simply typing OK.* However, in some situations, the break expression cannot be reset. For example, if a compiled function FOO incorrectly called putd and caused the error ARG NOT ATOM followed by a break on putd, EDIT might be able to find the form headed by FOO, and also find *that* form in some higher interpreted function. But after the user corrected the problem in the FOO-form, if any, he would still not have in any way informed EDIT what to do about the immediate problem or the incorrect call to putd. However, if FOO were *interpreted* EDIT would find the putd form itself, so that when the user corrected that form, EDIT could use the new corrected form to reset the break expression. The two cases are shown below:

```
ARG NOT ATOM
(FUM)
(PUTD BROKEN)
:EDIT
IN FIE...
(FOO X)
EDIT
*(2 (CAR X))
*OK
NOTE: BRKEXP NOT CHANGED
FIE
:?=
U = (FUM)
:(SETQ U (CAR U))
FUM
:OK
PUTD
```

```
ARG NOT ATOM
(PUTD BROKEN)
:EDIT
IN FOO...
(PUTD X)
EDIT
*(2 (CAR X))
*OK
FOO
:OK
PUTD
```

*Evaluating the new brkexp will involve reevaluating the form that caused the break, e.g. if (PUTD (QUOTE (FOO)) big-computation) were handled by EDIT, big-computation would be reevaluated.

IN? similar to EDIT, but just prints parent form, and superform, but does not call editor, e.g.

```
NON-NUMERIC ARG
NIL
(IPLUS BROKEN)
:IN?
FOO:    (IPLUS X Z)
```

Although EDIT and IN? were designed for error breaks, they can also be useful for user breaks. For example, if upon reaching a break on his function FOO, the user determines that there is a problem in the call to FOO, he can edit the calling form and reset the break expression with one operation by using EDIT. The following two protocol's, with and without the use of EDIT, illustrate this.

```
(FOO BROKEN)                                   (FOO BROKEN)
:?=                                            :?=
X = (A B C)                                    X = (A B C)
Y = D                                          Y = D
:BT                                            :EDIT
                                               IN FIE...
FOO                                            (FOO V U)
SETQ                                           EDIT
COND           find which function             *(SW 2 3)
PROG           FOO is called from              *OK
FIE                                            FIE              *
               (aborted with ↑E)               :OK
:EDITF(FIE)                                    FOO
EDIT
*F FOO P
(FOO V U)      edit it
*(SW 2 3)
*OK
FIE
:(SETQ Y X)    reset X and Y
(A B C)
:(SETQQ X D)
D
:?=
X = D
Y = (A B C)    check them
:OK
FOO
```

*x and y have not been changed, but brkexp has. See previous footnote.

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

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 treated as a function or variable as before.

Example: the command ARGS could be defined by including on breakmacros: (ARGS (PRINT (VARIABLES LASTPOS T)))

Break Functions

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

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, !GO, OK, !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. brktype is NIL for user breaks, INTERRUPT for control-H breaks, and ERRORX for error breaks.

For error breaks, the input buffer is cleared and saved. (For control-H breaks, the input buffer was cleared at the time the control-H was typed, see p. 16.3.) In both cases, if the break returns a value, i.e., is not aborted via ↑ or control-D, the input buffer will be restored (see p. 14.17).

breakØ[fn;when;coms]

sets up a break on the function fn by redefining fn as a call to breakl with brkexp an equivalent definition of fn, and when, fn, and coms, as brkwhen, brkfn, brkcoms. Puts property BROKEN on property list of fn with value a gensym defined with the original definition. Puts property BRKINFO on property list of fn with value (BREAKØ when coms). (For use in conjunction with rebreak.) Adds fn to the front of the list brokenfns. Value is fn.

If fn is non-atomic and of the form (fn1 IN fn2), breakØ 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 breakØ 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* or blockcompiled, whereas breakin only works on interpreted functions.

If fn1 is not found in fn2, breakØ 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, breakØ 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, _breakØ_ is called for each member of _fn_ using the same values for _when_, _coms_, and _file_ specified in this call to _breakØ_. This distributivity permits the user to specify complicated break conditions on several functions without excessive retyping, e.g.,

```
breakØ[(FOO1 ((PRINT PRIN1) IN
   (FOO2 FOO3)));(NEQ X T);
   (EVAL ?= (Y Z) OK)]
```

Will break on FOO1, PRINT-IN-FOO2, PRINT-IN-FOO3, PRIN1-IN-FOO2 and PRIN1-IN-FOO3.

If _fn_ is non-atomic, the value of _breakØ_ is a list of the 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) (EVAL))] is equivalent to breakØ[FOO1,T] and breakØ[FOO2; (GREATERP N 5); (EVAL)] trace[x]

is a nonspread nlambda. For each atomic argument, it performs breakØ[atom;T; (TRACE ?= NIL GO)]* For each list argument, car is the function to be traced, and cdr the forms the user wishes to see, i.e. trace 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 for FOO2. In the special case that the user wants to see *only* the value, he can perform TRACE((fn)). This sets up a break with commands (TRACE ?= (NIL) GO).

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

Breakin enables the user to insert a break, i.e. a call to breakl, at a specified location in an interpreted function. For example, if foo calls fie, inserting a break in foo before the call to fie is similar to breaking fie. However, breakin can be used to insert breaks before or after prog labels, particular SETQ expressions, or even the evaluation of a variable. This is because breakin operates by calling the editor and actually inserting a call to breakl at a specified point *inside* of the function.

The user specifies where the break is to be inserted by a sequence of editor commands. These commands are preceded by BEFORE, AFTER, or AROUND, which breakin uses to determine what to do once the editor has found the specified point, i.e. put the call to breakl BEFORE that point, AFTER that point, or AROUND that point. For example, (BEFORE COND) will insert a break before the first occurrence of cond, (AFTER COND 2 1) will insert a break after the predicate in the first cond clause, (AFTER BF (SETQ X &)) after the *last* place X is set. Note that (BEFORE TTY:) or (AFTER TTY:) permit the user to type in commands to the editor, locate the correct point, and verify it for himself using the P command, if he desires, and exit from the editor with OK.* breakin then inserts the break BEFORE, AFTER, or AROUND that point.

For breakin BEFORE or AFTER, the break expression is NIL, since the value of the break is usually not of interest. For breakin AROUND, the break expression will be the indicated form. When in the break, the user can use the EVAL command to evaluate that form, and examine its value, before allowing the computation to proceed. For example, if the user inserted a break after a cond

* A STOP command typed to TTY: produces the same effect as an unsuccessful edit command in the original specification, e.g., (BEFORE CONDD). In both cases, the editor aborts, and breakin types (NOT FOUND).

predicate, e.g. (AFTER (EQUAL X Y)), he would be powerless to alter the flow of computation if the predicate were not true, since the break would not be reached. However, by breaking (AROUND (EQUAL X Y)), he can evaluate the break expression, i.e. (EQUAL X Y), look at its value, and return something else if he wished.

The message typed for a breakin break, is ((fn) BROKEN), where fn is the name of the function inside of which the break was inserted. Any error, or typing control-E, will cause the full identifying message to be printed, e.g. (FOO BROKEN AFTER COND 2 1)

A special check is made to avoid inserting a break inside of an expression headed by any member of the list nobreaks, initialized to (GO QUOTE *), since this break would never be activated. For example, if (GO L) appears before the label L, breakin (AFTER L) will not insert the break inside of the GO expression, but skip this occurrence of L and go on to the next L, in this case the label L.

breakin[fn,where,when,coms]	<u>breakin</u> is an <u>nlambda</u>. <u>when</u> and <u>coms</u> are similar to <u>when</u> and <u>coms</u> for <u>breakØ</u>, except that 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. (See earlier discussion).

If <u>fn</u> is a compiled function, <u>breakin</u> returns (fn 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, <u>breakin</u> adds the property BROKEN-IN with value to T, and the property BRKINFO with value (where when coms) to the property list of <u>fn</u>, and adds <u>fn</u> to the front of the list <u>brokenfns</u>.

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

unbreak[x]

unbreak is a nospread nlambda. It takes an indefinite number of functions modified by break, trace, or breakin and restores them to their original state by calling unbreakØ. Value is list of values of unbreakØ.

unbreak[] will unbreak all functions on brokenfns, in reverse order. It first sets brkinfolst to NIL.

unbreak[T] unbreaks just the first function on brokenfns, i.e., the most recently broken function.

unbreakØ[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. Adds property value of BRKINFO to (front of) brkinfolst.

Note: unbreakØ[(fn1 IN fn2)] is allowed: unbreakØ 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. Value is fn. Unbreakin is called by unbreak if fn has property BROKEN-IN with value T on its property list.

rebreak[x] is an nlambda, nospread function for rebreaking functions that were previously broken without having to respecify the break information. For each function on x, rebreak searches brkinfolst for break(s) and performs the corresponding operation. Value is a list of values corresponding to calls to breakØ or breakin. If no information is found for a particular function, value is (fn - NO BREAK INFORMATION SAVED).

rebreak[] rebreaks everything on brkinfolst, i.e., rebreak[] is the inverse of unbreak[].

rebreak[T] rebreaks just the first break on brkinfolst, i.e., the function most recently unbroken.

changename[fn;from;to] changes all occurrences of from to to in fn. fn may be compiled or block-compiled. Value is fn if from was found, otherwise NIL. Does not perform any modifications of property lists. Note that from and to do not have to be functions, e.g.they can be names of variables.

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, i.e. it works on a copy. 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.

baktrace[pos1;pos2;skipfn;varsflg;*form*flg;allflg]

prints backtrace from pos1 to pos2. If skipfn is not NIL, and skipfn[stkname[pos]] is T, pos is skipped (including all variables).
varsflg=T for backtrace a la BTV
varsflg=T,*form*flg=T - BTV*
varsflg=T,allflg=T - BTV!

SECTION XVI

ERROR HANDLING

Contents

Error Handling in LISP

There are currently twenty-nine different error types in the BBN LISP system. These are discussed in greater detail later in this section. However, by far the most common "error conditions" in (interpreted) LISP programs, unbound atoms and undefined functions, are not treated as errors at all, but handled in a special way by the interpreter. The basic difference between real errors, and unbound atoms and undefined functions, is that real errors can (usually) occur inside of hand coded SUBRs, or in compiled code, where the stack may not be in a clean enough state to permit a function call. Therefore, the system must always 'back up' the stack to the last function call before it can call the error handling functions and allow the user to interact*. Unbound atoms and undefined functions, however, are detected by the interpreter when (before) it attempts to evaluate a LISP form. Consequently, the system is in a better position to allow the user to correct the unbound atom or undefined function and then continue without losing any of his computation.

*One exception is error type 1, UNDEFINED FUNCTION call from compiled code. For this type of error, nothing will be lost since the system does not have to 'back up' but can simply call the error handling functions instead of the (undefined) function it was getting ready to call.

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 exactly as though it were the value of the form.

faulteval is defined to print either U.B.A., for unbound atom, or U.D.F., for undefined function, and then to call break1 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.

The decision over whether or not to induce a break depends on the depth of computation, and the amount of time invested in the computation. The actual algorithm is described in detail below in the section on breakcheck. Suffice it to say that the parameters affecting this decision have been adjusted empirically so that trivial type-in errors do not cause breaks, but deep errors do.

*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.

Teletype Initiated Breaks

Control-H

Section XV on the break package described how the user could cause a break when a specified function was entered. The user can also indicate his desire to go into a break at any time while a program is running by typing control-H.* At the next point a function is about to be entered, the function interrupt is called instead.** interrupt types INTERRUPTED BEFORE followed by the function name, constructs an appropriate break expression, and then calls break1. The user can then examine the state of the computation, and continue by typing OK, GO or EVAL, and/or retfrom back to some previous point, exactly as with a user break. Control-H breaks are thus always 'safe'. Note that control-H breaks are not affected by the depth or time of the computation. However, they *only* occur when a function is called, since it is only at this time that the system is in a "clean" enough state to allow the user to interact. Thus, if a compiled program is looping without calling any functions, or is in a I/O wait, control-H will not affect it. Control-B, however, will.

Control-B

Control-B is a stronger interruption than control-H. It effectively generates an immediate error. This error is treated like any other error except that it *always* causes a break, regardless of the depth or time of the computation.*** Thus if the function FOO is looping internally, typing control-B will

---

* As soon as control-H is typed, LISP clears and saves the input buffer, and then rings the bell, indicating that it is now safe to type ahead to the upcoming break. If the break returns a value, i.e., is not aborted via ↑ or control-D, the contents of the input buffer before the control-H was typed will be restored, see p. 15.17.

** interrupt is also called for UNDEFINED FUNCTION calls from compiled code.

*** However, setting helpflag to NIL will suppress the break. See discussion of breakcheck below.

cause the computation to be stopped, the stack unwound to the point at which FOO was called, and then cause a break. Note that the internal variables of FOO are not available in this break, and similarly, FOO may have already produced some changes in the environment before the control-B was typed. Therefore whenever possible, it is better to use control-H instead of control-B.

Control-E

If the user wishes to *abort* a computation, without causing a break, he should type control-E. Control-E does not go through the normal error machinery of scanning the stack, calling breakcheck, printing a message, etc. as described below, but simply types a carriage return and unwinds.

"Real" Errors

Real errors are handled similarly to U.B.A. and U.D.F. errors, except that the stack must first be backed up to the last function call. Thus, 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. For example if the compiled function hypotenuse were defined as:

```
(HYPOTENUSE
  [LAMBDA (X Y)
    (EXPT (FPLUS (FTIMES X X)
                 (FTIMES Y Y))
          .5])
```

and the user performed:

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

NON-NUMERIC ARG
NIL

(HYPOTENUSE BROKEN)
:?=
X = 3
Y = NIL

:(SETQ Y 4)
4
:OK
HYPOTENUSE
2.5
```

the stack would be backed up to the point hypotenuse was called, since ftimes and fplus compile open (Section 18) and expt would not yet have been entered, the error occurring in the evaluation of one of its arguments. Thus some partial results of the computation would be lost when the error occurred, i.e., the result of (FTIMES X X). In this particular case, the user could proceed as shown.

Note that the computation might have made some changes in the program's environment before the error occurred. For example, if hypotenuse were defined instead as:

```
(HYPOTENUSE
  [LAMBDA (X Y)
    (EXPT (FPLUS (SETQ X (FTIMES X X))
                 (FTIMES Y Y))
          .5])

←(FTIMES (SINE 30) (HYPOTENUSE 3))

NON-NUMERIC ARG
NIL
(HYPOTENUSE BROKEN)
:?=
X = 9
Y = NIL

:RETURN (HYPOTENUSE 3 4)
HYPOTENUSE = 5
2.5
←
```

The user must evaluate each situation individually to decide what should be done.

Breakcheck - When to Break

The decision as to whether or not to induce a break when an error occurs (U.B.A., U.D.F., or any of 29 real errors) is handled by the function breakcheck.* The user can suppress all error breaks by setting the variable helpflag to NIL (initially set to T). If helpflag=T, the decision is affected by two factors: the length of time spent in the computation, and the depth of the computation at the time of the error.** If the time is greater than helptime or the depth is greater than helpdepth, breakcheck returns T, i.e., a break will occur.

Since a function is not actually entered until its arguments are evaluated,*** the depth of a computation is defined to be the sum of the number of function calls plus the number of internal calls to eval. Thus if the user types in the expression

```
(MAPC FOO (FUNCTION (LAMBDA (X)
   (COND
     ((NOT (MEMB X FIE)) (PRINT X]
```

for evaluation, and FIE is not bound, at the point of the U.B.A. FIE error, two functions, mapc and cond, have been entered, and there are three internal calls to eval corresponding to the evaluation of the forms (COND ((NOT (MEMB X FIE)) (PRINT X))) (NOT (MEMB X FIE)), and (MEMB X FIE).**** The depth is thus 5.

*Breakcheck is not actually available to the user for advising or breaking since the error package is block-compiled.

**Except that control-B errors always break.

***Unless of course the function does not have its arguments evaluated, i.e. is an FEXPR, FEXPR*, CFEXPR, CFEXPR*, FSUBR or FSUBR*.

****For complete discussion of the stack and the interpreter, see Section 12.

breakcheck begins by measuring the length of time spent in the computation by subtracting the value of (CLOCK 2)* from the variable helpclock, which is rebound to (CLOCK 2) for each type-in to evalqt or break. If the difference is greater than helptime milliseconds, initially set to 1000, then a break will occur, i.e., breakcheck returns T.

The time criterion for breaking can be suppressed by *setting* helptime to NIL (or a very big number), or by *binding* helpclock to NIL. Note that *setting* helpclock to NIL will not have any effect because helpclock is rebound in the evalqt loop and by break.

breakcheck continues by searching back up the parameter stack looking for an errorset.** At the same time, it counts the number of internal calls to eval, as indicated by pseudo-variable bindings called evalblips. See pp. 12.5-12.6. As soon as (if) the number of evalblips exceeds helpdepth, breakcheck can stop searching for errorset and return T, since the position of the errorset is only needed when a break is *not* going to occur. Otherwise, breakcheck continues searching until either an errorset is found or the top of the stack is reached.

If breakcheck has not been able to decide in favor of a break, i.e., has not yet returned T, it then completes the depth check by counting the number of function calls between the error and the last errorset, or the top of the stack. If the number of calls plus the number of evalblips (already counted) is greater than or equal

---

*Whose value is number of milliseconds of compute time. See section 21.

**errorsets are simply markers on the stack indicating how far back unwinding is to take place when an error occurs, i.e. they segment the stack into sections such that if an error occurs in any section, control returns to the point at which the last errorset was entered, from which NIL is returned as the value of the errorset. See p. 16.14.

to helpdepth, initially set to 9,* breakcheck returns T. Otherwise, it records the position of the last errorset, and the value of errorset's second argument, which is used in deciding whether to print the error message, and returns NIL.

If breakcheck is NIL, i.e., a break is *not* going to occur, then if an errorset was found, NIL is returned (via retfrom) as the value of the errorset, after first printing the error message if the errorset's second argument was TRUE. If there was no errorset, the message is printed, and the error routines 'reset', i.e., return to evalqt. This procedure is followed for all types of errors.

Note that for all error breaks, break1 will clear and save the input buffer. If the break returns a value, i.e., is not aborted via ↑ or control-D, the input buffer will be restored. See p. 15.17.

*Arrived at empirically, takes into account the overhead due to evalqt or break.

Error Types

There are currently twenty-nine error types in the BBN-LISP system. They are listed below by error number. This number is set internally by the code that detects the error, before it calls the error handling functions. It is also the value returned by _errorn_ after that type of error occurs, and is used by _errormess_ for printing the error message.

Most error types will print the offending expression following the message, e.g., NON-NUMERIC ARG NIL is very common. Error type 18, always causes a break (unless _helpflag_ is NIL). All other errors cause breaks if _breakcheck_ returns T.

0	NONXMEM	reference to non-existent memory. Usually indicates system is sick.
1	UNDEFINED FUNCTION	call from compiled code to an undefined function.
2	P-STACK OVERFLOW	occurs when computation is too deep, either with respect to number of function calls, or number of variable bindings. Usually because of a non-terminating recursive computation, i.e. a bug.
3	ILLEGAL RETURN	call to _return_ when not inside of an interpreted _prog_.
4	ILLEGAL ARG - PUTD	second argument to _putd_ (the definition) is not NIL, a list, or a pointer to compiled code.

5	ARG NOT ATOM - SET	first argument to set (name of the variable) not a literal atom.
6	ATTEMPT TO SET NIL	via set or setq
7	ATTEMPT TO RPLAC NIL	attempt either to rplaca or to rplacd NIL with something other than NIL
8	UNDEFINED OR ILLEGAL GO	go when not inside of a prog, or go to nonexistent label
9	FILE WON'T OPEN	From infile or outfile, see p. 14.2.
10	NON-NUMERIC ARG	a numeric function e.g. iplus, itimes, igreaterp, expected a number.
11	ATOM TOO LONG	≥ 100 characters
12	ATOM HASH TABLE FULL	no room for any more (new) atoms
13	FILE NOT OPEN	from an I/O function, e.g. read, print, closef.
14	ARG NOT ATOM	
15	TOO MANY FILES OPEN	≥ 8 including teletype.
16	END OF FILE	from an input function, e.g. read, readc, ratom. Note: file will then be closed.
17	ERROR	call to error.
18	BREAK	control-B was typed

19	ILLEGAL STACK ARG	a stack function expected a stack position and was given something elst. This might occur if the arguments to a stack function are reversed. Also occurs if user specified a stack position with a function name, and that function was not found on the stack. See section 12.
20	FAULT IN EVAL	artifact of bootstrap. Never occurs after faulteval has been defined as described earlier.
21	ARRAYS FULL	system will first initiate a GC: 1, and if no array space is reclaimed, will then generate this error.
22	DIRECTORY FULL	no new files can be created until user deletes some old ones and expunges.
23	FILE NOT FOUND	file name does not correspond to a file in the corresponding directory. Can also occur if file name is ambiguous.
24	FILE INCOMPATIBLE - SYSIN	from sysin, see p. 14.22.
25	UNUSUAL CDR ARG LIST	a form ends in a non-list other than NIL, e.g. (CONS T . 3)
26	HASH TABLE FULL	see hash link functions, section 7.
27	ILLEGAL ARG	Catch-all error. Currently used by evala, arg, funarg, allocate, rplstring, and sfptr.
28	ARG NOT ARRAY	elt or seta given an argument that is not a pointer to the beginning of an array
29	OVERFLOW/UNDERFLOW	see p. 13.3, 13.8

Error Functions

errorx[erxm] — is the entry to the error routines. If erxm=NIL, errorn[] is used to determine the error-message. Otherwise, seterrorn[erxm] is performed, 'setting' the error type and argument. Thus following either errorx[(10 T)] or (PLUS T), errorn[] is (10 T). errorx calls breakcheck, and either induces a break or prints the message and unwinds to the last errorset. Note that errorx can be called by any program to intentionally induce an error of any type. However, for most applications, the function error will be more useful.

error[mess1;mess2;nobreak] — prints mess1 (using prin1), followed by a space if mess1 is an atom, otherwise a carriage return, then prints mess2, using prin1 if mess2 is a string, otherwise print. e.g., error["NON-NUMERIC ARG";T] will print

```
NON-NUMERIC ARG
T
```

and error[FOO;"NOT A FUNCTION"] will print FOO NOT A FUNCTION. (If both mess1 and mess2 are NIL, error prints ERROR.) If nobreak=T, error then calls error!, otherwise it calls errorx[(17 (mess1 . mess2))], i.e. generates an error of type 17. The decision as to whether or not to break is then handled as per any other error.

help[mess1;mess2]	prints _mess1_ and _mess2_ a la _error_, and then calls _break1_. If both _mess1_ and _mess2_ are NIL, HELP! is used for the message. _help_ is a convenient way to program a default condition, or to terminate some portion of a program which theoretically the computation is never supposed to reach.
error![]*	programmable control-E, i.e., immediately returns from last _errorset_ or resets.
reset[]	Programmable control- D i.e. immediately returns to the top level.
errorn[]	returns information about the last error in the form (n x) where _n_ is the error type number and _x_ is the expression which was (would have been) printed out after the error message. Thus following (PLUS T), errorn[] is (10 T).
errormess[u]	prints message corresponding to an _errorn_ that yielded _u_. For example, errormess[(10 T)] would print NON-NUMERIC ARG T

*Pronounced "error-bang"

errorset[u;v]*

performs eval[u]. 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. In most cases, ersetq and nlsetq (described below) are more useful. If no error occurs in the evaluation of u, the value of errorset is a list containing one element, the value of eval[u]. If an error did occur, the value of errorset is NIL.

The argument v controls the printing of error messages if an error occurs. If v=T, the error message is printed; if v=NIL it is not.

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].

*errorset is a subr, so the names 'u' and 'v' don't actually appear on the stack nor will they affect the evaluation.

SECTION XVII

AUTOMATIC ERROR CORRECTION - THE DWIM FACILITY

Contents

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 of the LISP program or expression in question, e.g. misspellings, certain kinds of parentheses errors, etc. To correct these types of errors we have implemented in BBN-LISP a DWIM facility, short for Do-What-I-Mean. DWIM is called automatically whenever an error* occurs in the evaluation of a LISP expression. DWIM then proceeds to try to) correct the mistake using the current context of computation plus information about what the user had previously been doing, (and what mistakes he had been making) as guides to the remedy of the error. If DWIM is able to make the correction, the computation continues as though no error had occurred. Otherwise, the procedure is the same as though DWIM had not intervened: a break occurs, or an unwind to the last errorset, as described in Chapter 16. The following protocol illustrates the operation of DWIM.

*Currently, DWIM only operates on unbound atoms and undefined function errors.

Example

The user defines a function fact of one argument, n. The value of fact[n] is to be n factorial.

```
←DEFINEQ((FACT (LAMBDA (N) (COND
((ZEROP N9 1) ((T (ITIMS N (FACCT 8SUB1 N]
(FACT)
←
```

Note that the definition of fact contains several mistakes: Itimes 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; and finally, there is an extra left parenthesis in front of the T that begins the final clause in the conditional.

```
←PRETTYPRNT((FACCT]                                      [1]
=PRETTYPRINT                                             [2]
=FACT                                                    [3]

(FACT
  [LAMBDA (N)
    (COND
      ((ZEROP N9 1)
        ((T (ITIMS N (FACCT 8SUB1 N])
NIL
←
```

After defining fact, the user wishes to look at its definition using PRETTYPRINT, which he unfortunately misspells. [1] Since there is no function PRETTYPRNT in the system, a U.D.F. error occurs, and DWIM 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, notifies the user of this, [2] and calls prettyprint.

At this point, PRETTYPRINT would normally print (FACCT NOT PRINTABLE) and exit, since facct has no definition. Note that this is *not* a LISP error condition, so that DWIM would not be called as described above. However, it is obviously not what the user *meant*.

This sort of mistake is corrected by having prettyprint itself explicitly invoke the spelling corrector portion of DWIM whenever given a function with no expr definition. Thus with the aid of DWIM, prettyprint is able to determine that the user wants to see the definition of the function fact, [3] and proceeds accordingly.

```
←FACT(3)                                                        [4]
N9(IN FACT) >>--> N)
(IN FACT)  (COND -- ((T --))) >>--> (COND -- (T --))
ITIMS(IN FACT)->ITIMES                                          [5]
FACCT(IN FACT)->FACT
8SUB1(IN FACT) >>--> (SUB1
6
←PP FACT                                                        [6]

(FACT
  [LAMBDA (N)
    (COND
      ((ZEROP N)
        1)
      (T (ITIMES N (FACT (SUB1 N])
NIL
←
```

The user now calls his function fact.[4] During its execution, five errors occur, and DWIM is called five times. [5] At each point, the error is corrected, a message printed describing the action taken, and the computation allowed to continue as if no error had occurred. Following the last correction, 6 is printed, the value of fact(3). Finally, the user prettyprints the new, now correct, definition of fact. [6]

In this particular example, the user was shown operating in TRUSTING mode, which gives DWIM carte blanche for all corrections. The user can also operate in CAUTIOUS mode, in which case DWIM will inform him of intended corrections before they are made, and allow the user to approve or disapprove of them. For most corrections, if the user does not respond in a specified interval of time, DWIM automatically proceeds with the correction, so that the user need intervene only when he does not approve. Sample output is given below. Note that the user responded to the first, second, and fifth questions; DWIM responded for him on the third and fourth.

```
←FACT(3)
U.B.A. N9(IN FACT)    FIX?   YES                    [1]
N9(IN FACT) >>--> N)
U.D.F. T(IN FACT)   FIX?   YES                      [2]
(COND -- ((T --))) >>--> (COND -- (T --))
ITIMS(IN FACT)->ITIMES ?  ...YES                    [3]
FACCT(IN FACT)->FACT ?  ...YES                      [4]
U.B.A. 8SUB1(IN FACT)   FIX?   NO                   [5]
U.B.A.
(8SUB1 BROKEN)
:
```

We have put a great deal of effort into making DWIM 'smart', and experience with perhaps a dozen different users indicates we have been very successful; DWIM seldom fails to correct an error the user feels it should have, and almost never mistakenly corrects an error. However, it is important to note that even when DWIM *is* wrong, no harm is done:* since an error had occurred, the user would have had to intervene anyway if DWIM took no action. Thus, if DWIM mistakenly corrects an error, the user simply interrupts or aborts the computation, UNDOes the DWIM change using UNDO described in Section 22, and makes the correction he would have had to make without DWIM. It is this benign quality of DWIM that makes it a valuable part of BBN-LISP.

*Except perhaps if DWIM's correction mistakenly caused a destructive computation to be initiated, and information was lost before the user could interrupt. We have not yet had such an incident occur.

Interaction with DWIM

DWIM is enabled by performing either DWIM[C], for CAUTIOUS mode, or DWIM[T] for TRUSTING mode.* In addition to setting _dwimflg_ to T and redefining _faulteval_ as described on page 17.15, DWIM[C] sets _approveflg_ to T, while DWIM[T] sets _approveflg_ to NIL. The setting of _approveflg_ determines whether or not the user wishes to be asked for approval before a correction that will modify the definition of one of his functions. In CAUTIOUS mode, i.e. _approveflg_=T, DWIM will ask for approval; in TRUSTING mode, DWIM will not. Note that for corrections to expressions typed in by the user for immediate execution,** DWIM always acts as though _approveflg_ were NIL, i.e. no approval necessary. In either case, DWIM always informs the user of its action as described below.

Spelling Correction Protocol

The protocol used by DWIM for spelling corrections is as follows: If the correction occurs in type-in, print = followed by the correct spelling, followed by a carriage return, and then continue, e.g.

```
        user types:   ←(SETQ FOO (NCOCN FIE FUM))
        DWIM types:   =NCONC
```

If the correction does not occur in type-in, print the incorrect spelling, followed by (IN function-name), ->, and then the correct spelling, e.g. ITIMS(IN FACT)->ITIMES as shown on page 17.3.*** Then if _approveflg_=NIL, print a carriage return, make the correction

---

*BBN-LISP arrives with DWIM enabled in CAUTIOUS mode. DWIM can be disabled by executing DWIM[]. See p. 17.23

**Typed into _lispx_. _lispx_ is used by _evalqt_ and _break_, as well as for processing the editor's E command. Functions that call the spelling corrector directly, such as _editdefault_, p. 9.85, specify whether or not the correction is to be handled as type-in. For example, in the case of _editdefault_, commands typed directly to the editor are treated as type-in, so that corrections to them will never require approval. Commands given as an argument to the editor, or resulting from macro expansions, or from IF, LP, ORR commands etc. are not treated as type-in, and thus approval will be requested if _approveflg_=T.

***The appearance of -> is to call attention to the fact that the user's function will be or has been changed.

and continue. Otherwise, print a few spaces and a ? and then wait for approval.* The user then has six options. He can:

1. Type Y; DWIM types ES, and proceeds with the correction.
2. Type N; DWIM types O, and does not make the correction.
3. Type ↑; DWIM does not make the correction, and furthermore guarantees that the error will not cause a break. See p. 17.15
4. Type control-E; for error correction, this has the same effect as typing N.
5. Do nothing; in which case DWIM will wait a specified interval,** and if the user has not responded, DWIM will type ... followed by the default answer.***
6. Type space or carriage return; in which case DWIM will wait indefinitely. This option is intended for those cases where the user wants to think about his answer, and wants to insure that DWIM does not get 'impatient' and answer for him.

The procedure for spelling correction on other than LISP errors is analagous. If the correction is being handled as type in, DWIM prints = followed by the correct spelling, and returns it to the function that called DWIM, e.g. =FACT as shown on page 17.2 . Otherwise, DWIM prints the incorrect spelling, followed by =, followed by the correct spelling. Then if <u>approveflg</u>=NIL, DWIM prints a carriage-return and returns the correct spelling. Otherwise, DWIM prints a few spaces and a ? and then waits for approval. The user can then respond with Y, N, control-E, space, carriage return, or do nothing as described above.

---

*Whenever an interaction is about to take place and the user has typed ahead, DWIM types several bells to warn the user to stop typing, then clears and saves the input buffers, restoring them after the interaction is complete. Thus if the user has typed ahead before a DWIM interaction, DWIM will not confuse his type ahead with the answer to its question, nor will his type ahead be lost.

**Equal to <u>dwimwait</u> seconds. DWIM operates by dismissing for 500 milliseconds, then checking to see if anything has been typed. If not, it dismisses again, etc. until <u>dwimwait</u> seconds have elapsed. Thus, there will be a delay of at most ½ second before DWIM responds to the user's answer.

***The default is always YES unless otherwise stated.

Note that since the spelling corrector itself is not errorset protected, typing N and typing control-E may have different effects when the spelling corrector is called directly.* The former simply instructs the spelling corrector to return NIL, and lets the calling function decide what to do next; the latter causes an error which unwinds to the last errorset, however far back that may be.

Parentheses Errors Protocol

As illustrated earlier on page 17.3 , DWIM will correct errors consisting of typing 8 for left parenthesis and 9 for right parenthesis. In these cases, the interaction with the user is similar to that for spelling correction. If the error occurs in type-in, DWIM types = followed by the correction, e.g.

```
user types:    ←(SETQ FOO 8CONS FIE FUM]
DWIM types:    =(CONS
lispx types:   (A B C D)
```

Otherwise, if the error does not occur in type-in, there are two cases: approveflg=NIL and approveflg=T. If approveflg=NIL, i.e. no approval necessary, DWIM makes the correction and prints a message consisting of the offending atom, followed by (IN function-name), >>-->, the correction, and a carriage return, e.g. N9(IN FACT) >>--> N) as shown on page 17.4.

If approveflg=T, DWIM prints U.B.A. or U.D.F. followed by the offending atom, (IN function-name), several spaces, and then FIX? and waits for approval, e.g. U.B.A. N9(IN FACT) FIX? as shown on page 17.4. The user then has the same six options as for spelling correction (the default answer is NO). If the user types Y, DWIM then operates exactly the same as when approveflg=NIL, i.e. makes the correction and prints its message.

---

*The DWIM error correction routines *are* errorset protection.

U.D.F. T Errors Protocol

DWIM corrects certain types of parentheses errors involving a T clause in a conditional, namely errors of the form:

1. (COND --)(T --), i.e. the T clause appears outside, and immediately following the COND;
2. (COND --(-- & (T --))), i.e. the T clause appears inside a previous clause; and
3. (COND --((T --))), i.e. the T clause has an extra pair of parentheses around it.[†]

If the error occurs in type-in, DWIM simply types T FIXED and makes the correction. Otherwise if _approveflg_=NIL, DWIM makes the correction, and prints a message consisting of (IN function-name), followed by one of the above incorrect forms of COND, followed by >>-->, the corresponding correct form of the COND, and a carriage return, e.g. (IN FACT)(COND -- ((T --))) >>--> (COND -- (T --)) as shown on page 17.3.

If _approveflg_=T, DWIM prints U.D.F. T, followed by (IN function-name), several spaces, and then FIX? and waits for approval. The user then has the same options as for spelling corrections and parenthesis errors. If the user types Y or defaults, DWIM then proceeds exactly the same as when _approveflg_=NIL, i.e. makes the correction and prints its message, as shown on page 17.4.

[†]For U.D.F. T errors not of one of these three types, DWIM takes no corrective action at all, i.e. the error will occur.

Having made the correction, DWIM must then decide how to proceed with the computation. In case 1, (COND --)(T --), DWIM cannot know whether the last clause of the COND before the T clause succeeded or not, i.e. if the T clause had been inside of the COND, would it have been entered? Therefore DWIM asks the user 'CONTINUE WITH T CLAUSE' (with a default of YES). If the user types N, DWIM continues with the form after the COND, i.e. the form that originally followed the T clause.

In case 2, (COND -- (-- & (T --))), DWIM has a different problem. After moving the T clause to its proper place, DWIM must return as the value of the COND, the value of &. Since this value is no longer around, DWIM asks the user, 'OK TO REEVALUATE' and then prints &.† If the user types Y, or defaults, DWIM continues by reevaluating &, otherwise DWIM aborts, and a U.D.F. T error will then occur (even though the COND has in fact been fixed).

In case 3, (COND --((T --))), there is no problem with continuation, so no further interaction is necessary.

---

†In the special case where & is atomic, DWIM simply reevaluates it without asking approval.

Spelling Correction

The spelling corrector is given as arguments a misspelled word (word means literal atom), a spelling list (a list of words), and a number: xword, splst, and rel respectively. Its task is to find that word on splst which is closest to xword, in the sense described below. This word is called a *respelling* of xword. rel specifies the minimum 'closeness' between xword and a respelling. If the spelling corrector cannot find a word on splst closer to xword than rel, or if it finds two or more words equally close, its value is NIL, otherwise its value is the respelling.*

The exact algorithm for computing the spelling metric is described later on page 17.20, but briefly 'closeness' is inversely proportional to the number of disagreements between the two words, and directly proportional to the length of the longer word, e.g.PRTTYPRNT is 'closer' to PRETTYPRINT than CS is to CONS even though both pairs of words have the same number of disagreements. The spelling corrector operates by proceeding down splst, and computing the closeness between each word and xword, and keeping a list of those that are closest.** Certain differences between words are not counted as disagreements, for example a single transposition, e.g. CONS to CNOS, or a doubled letter, e.g. CONS to CONSS, etc. In the event that the spelling corrector finds a word on splst with *no* disagreements, it will stop searching and

---

*The spelling corrector can also be given an optional functional argument, fn, to be used for selecting out a subset of splst, i.e. only those members of splst that satisfy fn will be considered as possible respellings.

**The spelling corrector first checks for the special case that the first character in the xword is @ \ ↑ or ←, and replacing that character by the corresponding unshifted character, P, L, N, or O produces a word contained on splst or satisfying fn. In this case, that word will be the respelling, and the spelling list will not be searched. For example, if the user types @ACK, and the spelling corrector is called with fn = getd, PACK is *not* on splst, and in this case, PACK will be added to splst.

return this word as the respelling. Otherwise, the spelling corrector continues through the entire spelling list. Then if it has found one and only one 'closest' word, it returns this word as the respelling. For example, if xword is VONS, the spelling corrector will probably return CONS as the respelling. However, if xword is CONZ, the spelling corrector will not be able to return a respelling, since CONZ is equally close to both CONS and COND. If the spelling corrector finds an acceptable respelling it interacts with the user as described earlier.

In the special case that the misspelled word ends in alt-mode, the spelling corrector operates somewhat differently. Instead of trying to find the closest word as above, the spelling corrector searches for those words on splst that have the same initial characters as xword. In other words, it performs spelling *completion* instead of spelling correction. In this case, the entire spelling list is always searched, and if more than one respelling is found, the spelling corrector prints AMBIGUOUS, and returns NIL. For example, CON$ would be ambiguous if both CONS and COND were on the spelling list. If the spelling corrector finds one and only one respelling, it interacts with the user as described earlier.

For both spelling correction and spelling completion, regardless of whether or not the user approves of the spelling corrector's choice, the respelling is moved to the front of splst. Since many respellings are of the type with no disagreements, this procedure has the effect of considerably reducing the time required to correct the spelling of frequently misspelled words.

Spelling Lists

Although any list of atoms can be used as a spelling list, e.g. editcomsa, brokenfns, filelst, etc., four lists are maintained especially for spelling correction: spellingsl, spellings2, spellings3, and userwords.*

Spellingsl is a list of functions used for spelling correction when an input is typed in apply format, and the function is undefined, e.g. EDTIF(FOO). Spellingsl is initialized to contain defineq, break, makefile, editf, tcompl, load, etc. Whenever lispx is given an input in apply format, i.e. a function and arguments, the name of the function is added to spellingsl.** For example, typing ←CALLS(EDITF) will cause CALLS to be added to spellingsl. Thus if the user typed CALLS(EDITF) and later typed CALLLS(EDITV), since spellingsl would then contain CALLS, DWIM would be successful in correcting CALLLS to CALLS.†

Spellings2 is a list of functions used for spelling correction for all other undefined functions. It is initialized to contain functions such as addl, append, cond, cons, go, list, nconc, print, prog, return, setq, etc. Whenever lispx is given a non-atomic form, the name of the function is added to spellings2.**For example, typing (RETFROM (STKPOS (QUOTE FOO) 2)) to a break would add

---

*All of the remarks on maintaining spelling lists apply *only* when DWIM is enabled, as indicated by dwimflg=T.

**Only if the function is defined.

†If CALLLS(EDITV) were typed before CALLS had been 'seen' and added to spellingsl, the correction would not succeed. However, the alternative to using spelling lists is to search the entire oblist, a procedure that would make spelling correction intolerably slow.

retfrom to spellings2. Function names are also added to spellings2 by define, defineq, load (when loading compiled code), unsavedef, editf, and prettyprint.

Spellings3 is a list of words used for spelling correction on all unbound atoms. Spellings3 is initialized to editmacros, breakmacros, brokenfns, and advisedfns. Whenever lispx is given an atom to evaluate, the name of the atom is added to spellings3,* e.g. typing SPELLINGS1 to evalqt will add spellings1 to spellings3. Atoms are also added to spellings3 whenever they are edited by editv, and whenever they are set via rpaq or rpaqq. For example, when a file is loaded, all of the variables set in the file are added to spellings3. Atoms are also added to spellings3 when they are set by a lispx input, e.g. typing (SETQ FOO (REVERSE (SETQ FIE --))) will add both foo and fie to spellings3.

Userwords is a list containing both functions and variables that the user has *referred to* e.g. by breaking or editing. Userwords is used for spelling correction by arglist, unsavedef, prettyprint, break, editf, advise, etc. Userwords is initially NIL. Function names are added to it by define, defineq, load, (when loading compiled code, or loading exprs to property lists) unsavedef, editf, editv, editp, prettyprint, etc. Variable names are added to userwords at the same time as they are added to spellings3. In addition, the variable lastword is always set to the last word added to userwords, i.e. the last function or variable referred to by the user, and the respelling of NIL is defined to be the value of lastword. Thus, if the user has just defined a function, he can then edit it by simply typing EDITF(), or prettyprint it by typing PP().

*Only if the atom is bound

Each of the above four spelling lists are divided into two sections separated by a NIL. The first section contains the 'permanent' words; the second section contains the temporary words. New words are added to the corresponding spelling list at the front of its temporary section.* (If the word is already in the temporary section, it is moved to the front of that section; if the word is in the permanent section, no action is taken.) If the length of the temporary section then exceeds a specified number, the last (oldest) word in the temporary section is forgotten, i.e. deleted. This procedure prevents the spelling lists from becoming cluttered with unimportant words that are no longer being used, and thereby slowing down spelling correction time. Since the spelling corrector moves each word selected as a respelling to the front of its spelling list, the word is thereby moved into the permanent section. Thus once a word is misspelled and corrected, it is considered important and it will never be forgotten.

The maximum length of the temporary section for spellings1, spellings2, spellings3 and userwords is given by the value of #spellings1, #spellings2, #spellings3, and #userwords, initialized to 30, 30, 20, and 60 respectively. Using these values, the average length of time to search a spelling list for one word is about 4 milliseconds.[†]

---

*Except that functions added to spellings1 or spellings2 by lispx are always added to the end of the permanent section.

[†]If the word is at the front of the spelling list, the time required is only 1 millisecond. If the word is not on the spelling list, i.e. the entire list must be searched, the time is proportional to the length of the list; to search a spelling list of length 60 takes about 7 milliseconds.

Error Correction

As described on page 16.2, whenever the interpreter encounters an atomic form with no binding, or a non-atomic form car of which is not a function, it calls the function faulteval. When DWIM is enabled, faulteval is redefined to first call fixblock, a part of the DWIM package. If the user aborts by typing control-E, or if he indicates disapproval of DWIM's intended correction by answering N as described on p. 17.6, or if DWIM cannot decide how to fix the error, fixblock returns NIL.* In this case, faulteval proceeds exactly as described in Section 16, by printing a U.B.A. or U.D.F. message, and going into a break if the requirements of breakcheck are met, otherwise unwinding to the last errorset.

If DWIM can (and is allowed to) correct the error, fixblock exits by performing reteval of the corrected form, as of the position of the call to faulteval. Thus in the example at the beginning of the chapter, when DWIM determined that ITIMS was ITIMES misspelled, DWIM called reteval with (ITIMES N (FACCT 8SUB1 N)). Since the interpreter uses the value returned by faulteval exactly as though it were the value of the erroneous form, the computation will thus proceed exactly as though no error had occurred.

In addition to continuing the computation, DWIM also repairs the cause of the error whenever possible.** Thus in the above example, DWIM also changed (with rplaca) the expression (ITIMS N (FACCT 8SUB1 N)) that caused the error.

*If the user answers with ↑, (see p. 17.6) fixblock is exited by performing reteval[FAULTEVAL; (ERROR!)], i.e. an error is generated at the position of call to faulteval.

**If the user's program had *computed* the form and called eval, e.g. performed (EVAL (LIST X Y)) and the value of x was a misspelled function, it would not be possible to repair the cause of the error, although DWIM could correct the misspelling each time it occurred.

Error correction in DWIM is divided into three categories: unbound atoms, undefined cars of form, and undefined functions in apply. Assuming that the user approves if he is asked, the action taken by DWIM for the various types of errors in each of these categories is summarized below. The protocol of DWIM's interaction with the user has been described earlier.

Unbound Atoms

1. If the atom is an edit command (a member of editcomsa), and the error occurred in type-in, the effect is the same as though the user typed EDITF(), followed by the atom, i.e. DWIM assumes the user wants to be in the editor editing the last thing he referred to. Thus, if the user defines the function foo and then types P, he will see =FOO, followed by EDIT, followed by the printout associated with the execution of the P command, followed by *, at which point he can continue editing foo.

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. No message is typed, and no approval requested. If however the atom ends in alt-mode, DWIM will first perform spelling completion and then quote the respelling. The standard protocol for spelling corrections is followed.

   If the unbound atom is just ' itself, DWIM assumes the user wants the *next* expression quoted, e.g. (CONS X '(A B C)) will be changed to (CONS X (QUOTE (A B C))). Again no message will be printed or approval asked. (If no expression follows the ', DWIM gives up.)

3. If the atom contains an 8, DWIM assumes the 8 was intended to be a left parenthesis, and calls the editor to make appropriate repairs on the expression containing the atom. DWIM assumes that the user did not notice the mistake, i.e. that the entire expression was affected by the missing left parenthesis. For example, if the user types (SETQ X (LIST (CONS 8CAR Y) (CDR Z)) Y], the expression will be changed to (SETQ X (LIST (CONS (CAR Y)(CDR Z)) Y)).

   The 8 does not have to be the first character of the atom, e.g. DWIM will handle (CONS X8CAR Y) correctly.

4. If the atom contains a 9, DWIM assumes the 9 was intended to be a right parenthesis and operates as in number 3.

5. If the unbound atoms occurs in a function, DWIM attempts spelling correction using as a spelling list the list of lambda and prog variables of the function.

6. If the unbound atom occurred in a type-in to a break, DWIM attempts spelling correction using the lambda and prog variables of the broken function.

7. Otherwise, DWIM attempts spelling corrections using spellings3 and a functional argument specifying the variable have a value other than NOBIND.

If all fail, DWIM gives up.

Undefined car of Form

1. If car of the form is a small number, and the error occurred in type-in, DWIM assumes the form is really an edit command and operates as described in case 1 of unbound atoms.

2. If car of the form is T, DWIM assumes a misplaced T clause and operates as described on p. 17.8.

3. If car of the form is F/L, DWIM changes the F/L to FUNCTION(LAMBDA, e.g. (F/L (Y) (PRINT (CAR Y))) is changed to (FUNCTION (LAMBDA (Y) (PRINT (CAR Y]. No message is printed and no approval requested. If the user omits the variable list, DWIM supplies (X), e.g. (F/L (PRINT (CAR X] becomes (FUNCTION (LAMBDA (X) (PRINT (CAR X]. DWIM determines that the user has supplied the variable list when more than one expression follows F/L, car of the first expression is not a defined function, and every element in the first expression is atomic. For example, DWIM will supply (X) when correcting (F/L (PRINT (CDR X)) (PRINT (CAR X]. Note however that DWIM will make a mistake with (F/L (PRIN X) (LIST X)), thinking that (PRIN X) is the variable list.

4. If car of the form has an EXPR property, DWIM prints car of the form, followed by 'UNSAVED', performs an unsavedef, and continues. No approval is requested.

5. If car of the form is an edit command (a member of editcomsl), DWIM operates as in 1.

6. If car of the form contains an 8, DWIM assumes a left parenthesis was intended e.g. (CONS8CAR X).

7. If car of the form contains a 9, DWIM assures a right parenthesis was intended.

8. If the error occurs in a function, or in a type-in while in a break, DWIM checks to see if the last character in car of the form is one of the lambda or prog variables, and if the first n-1 characters are the name of a defined function, and if so makes the corresponding change, e.g. (MEMBERX Y) will be changed to (MEMBER X Y). The protocol followed will be the same as for that of spelling correction, e.g. if approveflg=T, DWIM will type MEMBERX(IN FOO)->MEMBER X?

9. Otherwise, DWIM attempts spelling correction using spellings2 as the spelling list, and getd as the optional functional argument.

If all fail, DWIM gives up.

Undefined Function in Apply

1. If the function is a number and the error occurred in type-in, DWIM assumes the function is an edit command, and operates as described in case 1 of unbound atoms, e.g. the user types (on one line) 3 -1 P.

2. If the function has an EXPR property, DWIM prints its name followed by 'UNSAVED' performs an unsavedef and continues. No approval is requested.

3. If the function is the name of an edit command (on either editcomsa or editcomsl), DWIM operates as in 1, e.g. user user types F COND P.

4. If the function name contains an 8, DWIM assumes a left parenthesis was intended, e.g. EDIT8FOO].

5. Otherwise DWIM attempts spelling correction using spellingsl as the spelling list, and getd as the optional functional argument.

If all fail, DWIM gives up.

Spelling Corrector Algorithm

The basic philosophy of DWIM spelling correction is to count the number of disagreements between two words, and use this number divided by the length of the longer of the two words as a measure of their relative disagreement. One minus this number is then the relative agreement or closeness. For example, CONS and CONX differ only in their last character. Such substitution errors count as one disagreement, so that the two words are in 75% agreement. Most calls to the spelling corrector specify rel=70,* so that a single substitution error is permitted in words of four characters or longer. However, spelling correction on shorter words is possible since certain types of differences such as single transpositions are not counted as disagreements. For example, AND and NAD have a relative agreement of 100.

The central function of the spelling corrector is chooz. chooz takes as arguments: a word, a spelling list, a minimum relative agreement, and an optional functional argument, xword, splst, rel, and fn respectively.[†]

chooz proceeds down splst examining each word. Words not satisfying fn or those obviously too long to be sufficiently close to xword are immediately rejected. For example, if rel=70, and xword is 5 characters long, words longer than 7 characters will be rejected.[††]

---

*Integers between 0 and 100 are used instead of numbers between 0 and 1 in order to avoid floating point arithmetic.

[†]fn = NIL is equivalent to fn=(LAMBDA NIL T).

[††]Words much *shorter* than xword cannot be rejected, since doubled letters are not counted as disagreements. For example, CONNSSS and CONS have a relative agreement of 100. (Certain teletype diseases actually produce this sort of stuttering.)

If tword, the current word on splst, is not rejected, chooz computes the number of disagreements between it and xword by calling a sub-function, skor.

skor operates by scanning both words from left to right one character at a time.[†] Characters are considered to agree if they are the same characters; or appear on the same teletype key, (i.e. a shift mistake), for example P agrees with @, * with : (and vice versa); or if the character in xword is a lower case version of the character in tword.[††] Characters that agree are discarded,[†††] and the skoring continues on the rest of the characters in xword and tword.

If the first character in xword and tword do *not* agree, skor checks to see if either character is the same as one previously encountered, and not accounted-for at that time. (In other words, transpositions are not handled by lookahead, but by *lookback*.) A displacement of two or fewer positions is counted as a transposition; a displacement by more than two positions is counted as a disagreement. In either case, both characters are now considered as accounted for and are discarded, and skoring continues.

[†] skor actually operates on the list of character codes for each word. This list is computed by chooz before calling skor using dchcon, so that no storage is used by the entire spelling correction process.

[††] Although model 33 teletypes do not have lower *case* characters (they do have lower *shift*), a not infrequent teletype malfunction is to transmit the lower case bit.

[†††] i.e. tword and xword are reset.

If the first character in _xword_ and _tword_ do not agree, and neither are equal to previously unaccounted-for characters, and _tword_ has more characters remaining than _xword_, _skor_ removes and saves the first character of _tword_, and continues by comparing the rest of _tword_ with _xword_ as described above. If _tword_ has the same or fewer characters remaining than _xword_, the procedure is the same except that the character is removed from _xword_.† In this case, a special check is first made to see if that character is equal to the *previous* character in _xword_, or to the *next* character in _xword_, i.e. a double character typo, and if so, the character is considered accounted-for, and not counted as a disagreement.††

When _skor_ has finished processing both _xword_ and _tword_ in this fashion, the value of _skor_ is the number of unaccounted-for characters, plus the number of disagreements, plus the number of transpositions, with two qualifications: (1) if both _xword_ and _tword_ have a character unaccounted-for in the same position, the two characters are counted only once, i.e. substitution errors count as only one disagreement, not two; and (2) if there are no unaccounted-for characters and no disagreements, transpositions are not counted. This permits spelling correction on very short words, such as edit commands, e.g. XRT->XTR.*

†Whenever more than two characters in either _xword_ or _tword_ are unaccounted for, _skoring_ is aborted, i.e. _xword_ and _tword_ are considered to disagree.

††In this case, the 'length' of xword is also decremented. Otherwise making _xword_ sufficiently long by adding double characters would make it be arbitrarily close to _tword_, e.g. XXXXXX would correct to PP.

*Transpositions are also not counted when _fastypeflg_=T, for example, IPULX and IPLUS will be in 80% agreement with _fastypeflg_=T, only 60% with _fastypeflg_=NIL. The rationale behind this is that transpositions are much more common for fast typists, and should not be counted as disagreements, whereas more deliberate typists are not as likely to combine transpositions and other mistakes in a single word, and therefore can use the more conservative metric. _fastypeflg_ is initially NIL.

DWIM Functions

dwim[x]	If x=NIL, disables DWIM; value is NIL. If x=C, enables DWIM in cautious mode; value is CAUTIOUS. If x=T, enables DWIM in trusting mode; value is TRUSTING. For all other values of x, generates an error.
addspell[x;splst;n]	Adds x to one of the four spelling lists as follows:* if splst=NIL, adds x to userwords and to spellings2. Used by defineq. if splst=0, adds x to userwords. Used by load when loading exprs to property lists. if splst=1, adds x to spellings1 (at end of permanment section). Used by lispx. if splst=2, adds x to spellings2 (at end of permanent section). Used by lispx. if splst=3, adds x to userwords and spellings3.

splst can also be a spelling list, in which case n is the (optional) length of the temporary section.

addspell sets lastword to x when splst=NIL, 0, or 3.

If x is not a literal atom, addspell takes no action.

*If x is already on the spelling list, and in its temporary section, addspell moves x to the front of that section. See p. 17.14 for complete description of algorithm for maintaining spelling lists.

misspelled?[xword;rel;splst;fn;flg]

If xword=NIL, misspelled? prints = followed by the value of lastword, and returns this as the respelling, without asking for approval. Otherwise, misspelled? checks to see if xword is really misspelled, i.e. if fn applied to xword is true, or xword is already contained on splst. In this case, misspelled? simply returns xword. Otherwise misspelled? computes and returns fixspell[word;rel;splst;fn;flg].

fixspell[xword;rel;splst;fn;flg]†

The value of fixspell is either the respelling of xword or NIL. fixspell performs all of the interactions described earlier, including requesting user approval if necessary.

If flg=NIL, the correction is handled in type-in mode, i.e. approval is never requested, and word is not typed. If flg=T, xword is typed (before the =) and approval is requested if approveflg=T.

The time required for a call to fixspell with a spelling list of length 60 when the entire list must be searched is .5 seconds. If fixspell determines that the first word on the spelling list is the respelling and does not need to search any further, the time required is .02 seconds. In other words, the time required is proportional to the number of words with which xword is compared, with the time for one comparison, i.e. one call to skor, being roughly .01 seconds (varies slightly with the number of characters in the words being compared.)

---

† fixspell has a sixth argument, lst, for internal use by DWIM.

The function chooz is provided for users desiring spelling correction without any output or interaction:

chooz[xword;splst;rel;fn;tieflg] The value of chooz is the corrected spelling of xword[†] or NIL. chooz performs no interaction and no output. If tieflg=T and a tie occurs, i.e. more than one word on splst is found with the same closeness, chooz returns the first word. If tieflg=NIL, and a tie occurs, chooz returns NIL.

fncheck[fn;nomessflg;spellflg] The task of fncheck is to check whether fn is the name of a function, and if not, to correct its spelling.* If fn is the name of a function or spelling correction is successful, fncheck adds the (corrected) name of the function to userwords using addspell, and returns it as its value.

nomessflg informs fncheck whether or not the calling function wants to handle the unsuccessful case: if nomessflg is T, fncheck simply returns NIL, otherwise it prints fn NOT A FUNCTION and generates a non-breaking error.

[†] chooz does not perform spelling *completion*, only spelling correction.

*Since fncheck is called by many low level functions such as arglist, unsavedef, etc., spelling correction only takes place when dwimflg=T, so that these functions can operate in a small LISP system which does not contain DWIM.

The definition of fncheck is simply:

```
(FNCHECK
  [LAMBDA (FN NOMESSFLG SPELLFLG)
    (PROG (X)
          (COND
            ((NOT (LITATOM FN))
              (GO ERROR))
            ((GETD FN))
            ([AND DWIMFLG
                  (CAR (NLSETQ (SETQ X (MISSPELLED?
                                         FN 70 USERWORDS
                                         (FUNCTION GETD)
                                         SPELLFLG]
              (SETQ FN X))
            (T (GO ERROR)))
          (AND DWIMFLG (ADDSPELL FN 0))
          (RETURN FN)
      ERROR
          (COND
            (NOMESSFLG (RETURN NIL)))
          (ERROR FN (QUOTE "NOT A FUNCTION")
                 T])
```

fncheck is currently used by arglist, unsavedef, prettyprint, break0, breakin, chngnm, advise, printstructure, firstfn, lastfn, calls, and edita. For example, break0 calls fncheck with nomessflg=T since if fncheck cannot produce a function, break0 wants to define a dummy one. printstructure however calls fncheck with nomessflg=NIL, since it cannot operate without a function.

Many other system functions call misspelled? or fixspell directly. For example, breakl calls fixspell on unrecognized atomic inputs before attempting to evaluate them, using as a spelling list a list of all break commands. Similarly, lispx calls fixspell on atomic inputs using a list of all lispx commands. When unbreak is given the name of a function that is not broken, it calls fixspell with two different spelling lists, first with brokenfns, and if that fails, with userwords. makefile calls misspelled? using filelst as a spelling list. Finally, load, bcompl, brecompile, tcompl, and recompile all call misspelled? if their input file(s) won't open.

SECTION XVIII

THE COMPILER AND ASSEMBLER

Contents

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 via ASSEMBLE.

The most common way to use the compiler is to compile from a symbolic (prettydef) file, producing a corresponding file which contains a 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 ST or F, 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

N or 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 the file CFACT for subsequent loading.

Compiler Questions

The compiler uses the free variables lapflg, strf, svflg, 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 are 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:

1. 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. If the answer is affirmative, compset will type FILE: to allow the user to indicate where the output is to be written.

There are three other possible answers to LISTING? — each of which specifies a complete mode for compiling. They are:

S	Same as last setting.
F	Compile to File (no definition of functions).
ST	STore new definitions.

* The LAP and LAP2 code is usually not of interest to the user.

Implicit in these three are the answers to the questions on disposition of compiled code and expr's, so questions 2 and 3 would not be asked if 1 were answered with S, F, or ST.

2. REDEFINE?

YES Causes each function to be redefined as it is compiled. The compiled code is stored and the function definition changed. The variable <u>strf</u> is set to T.

NO Causes function definitions to remain unchanged. The variable <u>strf</u> is set to NIL.

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

3. SAVE EXPRS?

If answered YES, <u>svflg</u> is set to T, and the exprs are saved on the property list of the function name. Otherwise they are discarded. The answer ST for the first question implies YES for this question, F implies NO, and S makes no change.

4. OUTPUT FILE:

If the compiled definitions are to be written for later loading, you should provide the name of a file on which you wish to save the code that is generated. If you answer T or TTY:, the output will be typed on the teletype (not particularly useful). If you answer N, NO, or NIL, output will *not* be done. If the file named is already open, it will continue to be used. The free variable <u>lcfil</u> is set to the name of the file.

Nlambdas

When compiling the call to a function, the compiler must prepare the arguments to the function 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 first look at the function definition cell of the called function. If the function is not defined, the compiler will then look for a definition among the functions in the file that is being compiled. If the function is not contained there, in the absence of any other information, the compiler will assume the function is of type 1. 'Other information' can be supplied by the user by including nlambda nospread functions on the list nlama (for nlambda atoms), and including nlambda spread functions on the list nlaml (for nlambda list). In other words, if there are type 2 or 3 functions called from the functions being compiled, and they are only defined in a separate file, they must be included on nlama or nlaml, or the compiler will incorrectly assume that their arguments are to be evaluated, and compile the calling function correspondingly. Note that this is only necessary if the compiler does not 'know' about the function. If the function is defined at compile time, or is contained in the same DEFINEQ as the functions that call it, or was compiled earlier in this file or another file, the compiler will automatically handle calls to that function correctly. nlama and nlaml are consulted *only* as a last resort, when the compiler has no information about the function in question.

Globalvars

Another top level free variable that affects compilations is globalvars. Any variables that appear on this list, and are used freely in a compiled function, are always accessed through their value cell. In other words, a reference to the value of this variable is equivalent to (CAR (QUOTE variable)), regardless of whether or not it appears on the stack, i.e., the stack is not even searched for this variable when the compiled function is entered. Similarly, (SETQ variable value) is equivalent to (RPLACA (QUOTE variable) value); i.e., it sets the top-level value. globalvars is initialized to a fairly large list of system variables, e.g., brokenfns, editmacros, #rpars, dwimflg, et al.

Standard compilations are also affected by the setting of linkfns and nolinkfns, although these are intended primarily for use in conjunction with block compilations. See "Linked function calls", p. 18.20.

Compiler Functions

Note: when a function is compiled from its in core definition, i.e., via compile (and certain calls to recompile), as opposed to tcompl (which uses the definitions on a file), and the function has been modified by break, trace, breakin, or advise, it is restored to its original state, and a message printed out, e.g., FOO UNBROKEN. Then, if the function is not defined as an expr, its property list is searched for the property EXPR (see savedef, section 8). If there is a property EXPR, its value is used for the compilation, otherwise, the compiler prints (fn NOT COMPILABLE), and goes on to the next function.

compile[x;flg]

x is a list of functions (if atomic, list[x] is used). compile first asks the standard compiler questions, and then compiles each function on x, using its in-core definition. Value is x.

If compiled definitions are being dumped to a file, the file is closed unless flg=T.

compile2[name;def]

compiles def, redefining name if strf=T.* compile2 is used by compile, tcompl, and recompile.

tcompl[files]

tcompl is used to 'compile files', i.e., given a symbolic load file (e.g., one created by prettydef), it produces a file that contains the same S-expressions as the original symbolic file, except that every defineq is replaced by the corresponding compiled definitions. This 'compiled' file can be loaded into any LISP system with load.

*strf is one of the variables set by compset, described earlier.

files is a list of symbolic files to be compiled (if atomic, list[files] is used). tcompl asks the standard compiler questions, except for OUTPUT FILE: Instead, the output from the compilation of each symbolic file is written on a file of the same name suffixed with COM, e.g., tcompl[(SYM1 SYM2)] produces two files, SYM1.COM and SYM2.COM.*

tcompl processes each file one at a time, reading successive S-expressions, and writing them onto the output file, unless they begin with DEFINEQ or DECLARE. For each DEFINEQ, tcompl adds any NLAMBDA's in the DEFINEQ to nlama or nlaml,** so that calls to the NLAMBDA's will be compiled correctly even if the functions are currently not defined. tcompl then compiles each function in the DEFINEQ. Expressions beginning with DECLARE are used to set up MACROs for the compilation. tcompl evaluates each expression in (cdr of) the DECLARE, presumably for effect.*** Note that the DECLARE must precede (in the file) any DEFINEQ's it is to affect.

---

* The file name is constructed from the name field only, e.g. TCOMPL[FOO.TEM;3] produces FOO.COM. The version number will be the standard default case.

** described earlier, p. 18.5.

***DECLARE is *defined* the same as QUOTE, so it will have no effect when the prettydefed file is loaded.

The value of tcompl is a list of the names of the output files. All files are properly terminated and closed.

Recompile

The purpose of recompile is to allow the user to update a compiled file without necessitating recompiling every function in the file. recompile does this by using the results of a previous compilation. It produces a compiled file identical to one that would have been produced by tcompl, but at a considerable savings in time by compiling selected functions and copying from an earlier tcompl or recompile file the compiled definitions for the remainder of t ıe functions in the file. Even more savings can be achieved if the symbolic file being recompiled is currently in-core, i.e., was previously loaded, or was made from the user's current system. In this case, recompile will not have to read in the file, but can work from the in-core definitions.*

If the functions to be recompiled are currently defined as exprs, then recompile can be called with just one argument, the symbolic file; the rest of the arguments will be set appropriately. In other words, the most common usage of recompile is in the following sequence, load[file;PROP], edit some functions (thus unsavedefing them), makefile[file], and recompile[file], producing a new compiled file exactly equivalent to tcompl[file]. The rest of the discussion of recompile explains nonstandard usages, e.g., the symbolic file has not been loaded, some of the functions that have been changed are currently not unsaved, etc.

---

*This requires that the user observe the conventions of the 'file package' described in chapter 14 when making the symbolic file, i.e., he used makefile or else used prettydef with arguments of the form fileFNS, file, and fileVARS.

recompile[pfile;cfile;fns;coreflg] _pfile_ is the name of the pretty _file_ to be compiled, _cfile_ is the name of the compiled _file_ containing compiled definitions that may be copied. _fns_ is a list of the functions in _pfile_ that are to be recompiled, i.e., they have been changed (or defined for the first time) since _cfile_ was made. Note that _pfile_, not _fns_, drives _recompile_, so that extra functions may appear on _fns_. If _fns_=T, all functions in _pfile_ currently defined as exprs (after unbreaking and unadvising) are recompiled.

recompile asks the standard compiler questions, except for OUTPUT FILE: As with _tcompl_, the output automatically goes to _pfile.COM_.* _recompile_ proceeds through _pfile_, reading each expression, and writing it on _pfile.COM_ unless it is a DECLARE or DEFINEQ. DECLAREs are evaluated as with _tcompl_. For each DEFINEQ, any NLAMBDAs are added to _nlama_ and _nlaml_, and then each function is compiled if it appears on _fns_, or _fns_=T and the function is an _expr_. Otherwise,

* In general, all constructions of the form pfile.COM, pfileFNS, pfileVARS, and pfileBLOCKS are performed using the name field only. For example, if _pfile_=FOO.TEM;3, pfile.COM means FOO.COM, pfileFNS means FOOFNS, etc.

recompile reads from cfile until it finds the compiled version of the function it is working on, and then copies it (and all compiler generated sub-functions) to pfile.COM.

Note that the user can thus modify an old compiled file so as to add new functions by prettydefing them in pfile and then including them on fns. Similarly, he can delete functions by simply not prettydefing them, since if they do not appear in pfile, they will never be compiled or copied to pfile.COM. Note, however that the entire process depends on the order of those functions in cfile that are to be copied being the same as those in pfile. For example, if FOO appears before FIE in cfile, but the order is reversed in pfile, then when recompile attempts to copy FIE, it will skip over FOO. Then when it attempts to copy FOO, it will read to the end of cfile and not find it. In this case, it will generate an error.

If the file pfile is in core, i.e., has been loaded, or else was prettydefed from this system, the user can take advantage of this by calling recompile with coreflg=T. In this case, the procedure is the same as described above, but recompile 'fakes' reading pfile, instead determining what is on pfile from pfilefns and pfilevars* (recompile does read the date from pfile, which it copies to the output file.)

---

* See footnote p. 18.10.

recompile will work correctly even for functions written via the third argument to prettydef using a FNS command. (See section 14).

If cfile=NIL, pfile.COM is used for copying from,* coreflg is set to T, and if fns is NIL, it too is set to T. This is the most common usage.

The value of recompile is the new compiled file, pfile.COM.

Open Functions

When a function is called from a compiled function, a system routine is invoked that sets up the parameter and control push lists as necessary for variable bindings and return information. As a result, function calls can take up to a millisecond per call. If the amount of time spent *inside* the function is small, this function calling time will be a significant percentage of the total time required to use the function. Therefore, many 'small' functions, e.g., car, cdr, eq, not, cons are always compiled 'open', i.e., they do not result in a function call. Other larger functions such as prog, selectq, mapc, etc. are compiled open because they are frequently used. It is useful to know exactly which functions are compiled open in order to determine where a program is spending its time. Therefore below is a list of those functions which when compiled do not result in function calls. Note that the next section, "Affecting the Compiled Code", tells how the user can make other functions compile open via MACRO definitions.

* In other words, if cfile, the file used for obtaining compiled definitions to be copied, is NIL, pfile.COM is used, i.e., same name as output but a different version number (one less) than the output file.

List of Functions that Compile Open

*
AC
ADD1
AND
ARG
ARRAYP
ASSEMBLE
ATOM
CAR
CDR
CAAR
 ETC.
CDDDAR
CDDDDR
CLOSER
COND
CONS
EQ
ERSETQ
FASSOC
FDIFFERENCE
FGTP
FIX
FIXP
FLAST
FLENGTH
FLOAT
FLOATP
FMEMB
FNTH
FPLUS
FQUOTIENT
FRPLACA
FRPLACD
FTIMES
FUNCTION
GO
IDIFFERENCE
IGREATERP
ILESSP
IMINUS
IPLUS
IQUOTIENT
IREMAINDER
ITIMES
LIST
LISTP
LITATOM
LOC
LOGAND
LOGOR
LSH
MAP
MAPC
MAPCAR
MAPCONC
MINUSP
NEQ
NLISTP
NLSETQ
NOT
NULL
NUMBERP
OPENR
OR
PROG
PROG1
PROGN
RETURN
RSH
SELECTQ
SETARG
SETN
SETQ
SMALLP
SOME
STRINGP
SUB1
VAG
ZEROP

Affecting the Compiled Code

The BBN LISP compiler includes a macro capability by which the user can affect the compiled code. Macros are defined by placing the macro definition on the property list of the corresponding function under the property MACRO. When the compiler begins compiling a form, it retrieves a macro definition for car of the form, if any, and uses it to direct the compilation.* The three different types of macro definitions are given below.

(1) Open macros - (LAMBDA...) or (NLAMBDA...)

A function can be made to compile open by giving it a macro definition of the form (LAMBDA...) or (NLAMBDA...), e.g., (LAMBDA (X) (COND ((GREATERP X Ø) X) (T (MINUS X)))) for abs. The effect is the same as though the macro definition were written in place of the function wherever it appears in a function being compiled, i.e., it compiles as an open LAMBDA or NLAMBDA expression. This saves the time necessary to call the function at the price of more compiled code generated.

(2) Computed macros - (atom expression)

A macro definition beginning with an atom other than LAMBDA, NLAMBDA, or NIL allows *computation* of the LISP expression that is to be compiled in place of the form. The atom which starts the macro definition is bound to cdr of the form being compiled. The expression following the atom is then evaluated, and the result of this evaluation is compiled in place of the form. For example, list could be compiled this way by giving it the macro definition:

* The compiler has built into it how to compile certain basic functions such as car, prog, etc., so that these will not be affected by macro definitions. These functions are listed on p. 18.13. However, some of them are themselves implemented via macros, so that the user could change the way they compile.

```
                    [X (LIST (QUOTE CONS)
                             (CAR X)
                             (AND (CDR X)
                                  (CONS (QUOTE LIST)
                                        (CDR X]
```

This would cause (LIST X Y Z) to compile as (CONS X (CONS Y (CONS Z NIL))). Note the recursion in the macro expansion.* ersetq, nlsetq, map, mapc, mapcar, mapconc, and some, are compiled via macro definitions of this type.

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

(3) Substitution macro - (NIL expression) or (list expression)

Each argument in the form being compiled is substituted for the corresponding atom in car of the macro definition, and the result of the substitution is compiled instead of the form, i.e., (SUBPAIR (CAR macrodef) (CDR form) (CADR macrodef)). For example, the macro definition of add1 is ((X) (IPLUS X 1)). Thus, (ADD1 (CAR Y)) is compiled as (IPLUS (CAR Y) 1). The functions

add1, sub1, neq, nlistp, zerop, flength, fmemb, fassoc, flast, and fnth

* list is actually compiled more efficiently

are all compiled open using substitution macros. Note that abs could be compiled open as shown earlier or via a substitution macro. A substitution macro, however, would cause (ABS (FOO X)) to compile as
(COND ((GREATERP (FOO X) Ø) (FOO X)) (T (MINUS (FOO X))))
and consequently (FOO X) would be evaluated three times.

Function and Functional Arguments

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 gensym 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.

Note that a considerable savings in time could be achieved by defining FOO1 as a computed macro of the form

```
(Z (LIST (SUBST (CADADR Z) (QUOTE FN) def)
         (CAR Z)))
```

where def is the definition of FOO1 as a function of just its first argument and FN is the name used for its functional argument in its definition. The expression compiled contains what was previously the functional argument to FOO1, as an open LAMBDA expression. Thus you save not only the function call to FOO1, but also 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.

---

* or an appropriate funarg expression, see section 11.

Block Compiling

Block compiling provides a way of compiling several functions into a single block. Function calls between the component functions of the block are very fast, and the price of using a free variable, namely the time required to look up its value on the stack, is paid only once — when the block is entered. Thus, compiling a block consisting of just a single recursive function may yield great savings if the function calls itself many times. e.g., equal, copy, and count are block compiled in BBN LISP.

The output of a block compilation is a single, usually large, function. This function looks like any other compiled function; it can be broken, advised, printstructured, etc. Calls from within the block to functions outside of the block look like regular function calls, except that they are usually linked (described below). A block can be entered via several different functions, called entries. These must be specified when the block is compiled.* For example, the error block has three entries, errorx, interrupt, and faultl. Similarly, the compiler block has nine entries.

---

* Actually the block is entered the same as every other function, i.e., at the top. However, the entry functions call the main block with their name as one of its arguments, and the block dispatches on the name, and jumps to the portion of the block corresponding to that entry point. The effect is thus the same as though there were several different entry points.

Specvars

One savings in block compiled functions results from not having to store on the stack the names of the variables bound within the block, since the block functions all 'know' where the variables are stored. However, if a variable bound inside of a block is to be referenced *outside* the block, it must be included on the list specvars. For example, helpclock is on specvars, since it is rebound inside of lispxblock and editblock, but the error functions must be able to obtain its latest value.

Retfns

Another savings in block compilation arrives from not storing on the stack the *names* of internal calls between functions inside of the block. However, if a function's name must be visible on the stack, e.g., if the function is to be returned from by retfrom, it must be included on the list retfns.

Blkapplyfns

Normally, a call to apply from inside a block would be the same as a call to any other function outside of the block. If the first argument to apply turned out to be one of the entries to the block, the block would have to be reentered. blkapplyfns enables a program to compute the name of a function in the block to be called next, without the overhead of leaving the block and reentering it. This is done by including on the list blkapplyfns those functions which will be called in this fashion, and by using blkapply in place of apply. For example, the calls to the functions RI, RO, LI, LO, BI, and BO in the editor are handled this way. If blkapply is given a function not on blkapplyfns, the effect is the same as a call to apply and no error is generated. Note however, that blkapplyfns must be set at *compile* time, not run time, and furthermore, that all functions on blkapplyfns must be in the block, or an error is generated (at compile time).

Blklibrary

Compiling a function open via a macro provides a way of eliminating a function call. For block compiling, the same effect can be achieved by including the function in the block. A further advantage is that the code for this function will appear only once in the block, whereas when a function is compiled open, its code appears at each place where it is called. Also, note that recursive functions cannot be compiled open via macros.

The block library feature provides a convenient way of including functions in a block. It is just a convenience since the user can always achieve the same effect by specifying the function(s) in question as one of the block functions, provided it has an expr definition at compile time. The block library feature simply eliminates the burden of supplying this definition.

To use the block library feature, place the names of the functions of interest on the list blklibrary, and their expr definition on the property list of the function under the property BLKLIBRARYDEF.* When the block compiler compiles a form, it first checks to see if the function being called is one of the block functions. If not, and the function is on blklibrary, its definition is obtained from the property value of BLKLIBRARYDEF, and it is automatically included as part of the block. For example, blklibrary includes length and nth when the edit block is compiled.

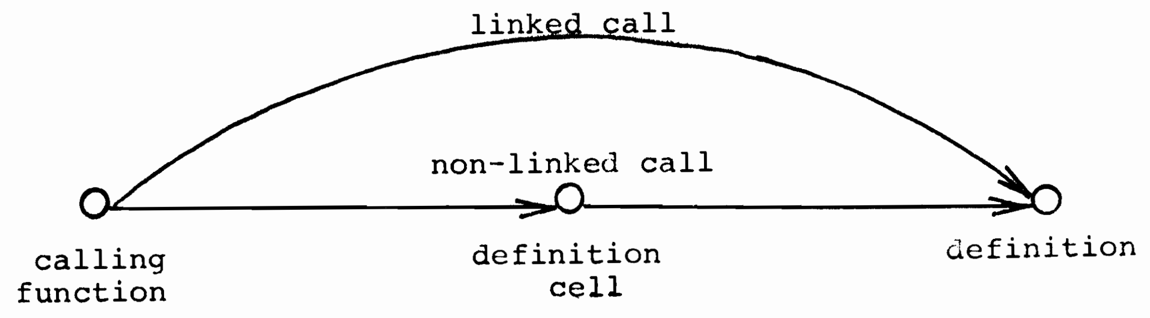
* The functions memb, assoc, equal, last, length, and nth already have BLKLIBRARYDEF properties.

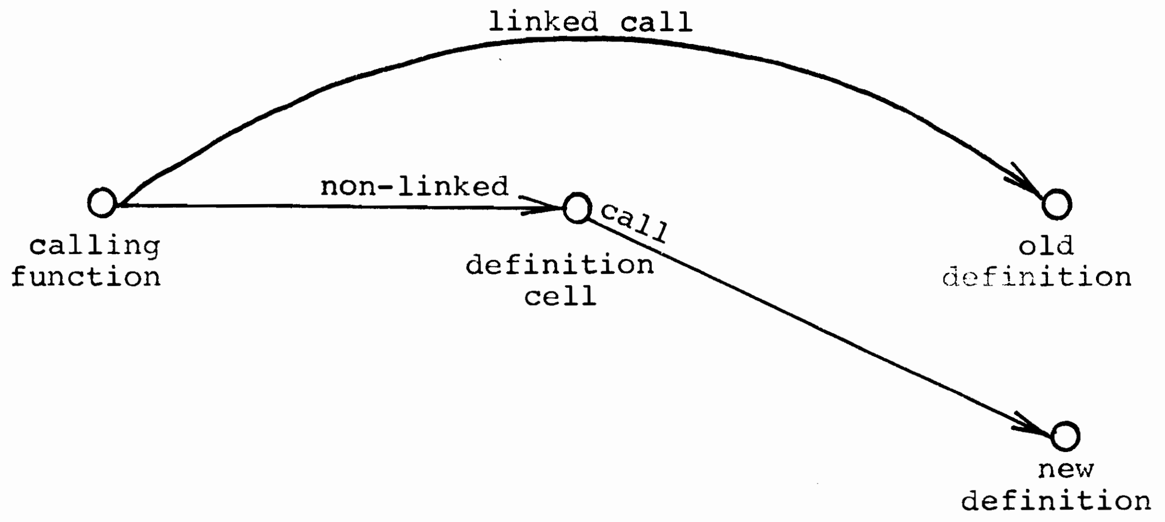
Linked Function Calls

Conventional (non-linked) function calls from a compiled function go through the function definition cell, i.e., the definition of the called function is obtained from its function definition cell at call time. Thus, when the user breaks, advises , or otherwise modifies the definition of the function FOO, every function that subsequently calls it instead calls the modified function. For calls from the system functions, this is clearly *not* a feature. For example, the user may wish to break on basic functions such as print, eval, rplaca, etc., which are used by the break package. In other words, we would like to guarantee that the system packages will survive through user modification (or destruction) of basic functions (unless the user specifically requests that the system packages also be modified). This protection is achieved by linked function calls.

For linked function calls, the definition of the called function is obtained at *link* time, i.e., when the calling function is defined, and stored in the literal table of the calling function. At *call* time, this definition is retrieved from where it was stored in the literal table, *not* from the function definition cell of the called function as it is for non-linked calls. These two different types of calls are illustrated in the figure on page 18.21.

Note that while function calls from block compiled functions are *usually* linked, and those from standardly compiled functions are *usually* non-linked, linking function calls and blockcompiling are independent features of the BBN LISP compiler, i.e., linked function calls are possible, and frequently employed, from standardly compiled functions.





Linked vs. Nonlinked Function Calls

Note that normal function calls require only the called function's name in the literals of the compiled code, whereas a linked function call uses two literals and hence produces slightly larger compiled functions.

The decision as to whether to link a particular function call is determined by the variables linkfns and nolinkfns as follows:

(1) If the function appears on nolinkfns, the call is not linked;
(2) If block compiling and the function is one of the block functions, the call is internal as described earlier;
(3) If the function appears on linkfns, the call is linked;
(4) If nolinkfns=T, the call is not linked;
(5) If block compiling, the call is linked;
(6) If linkfns=T, the call is linked;
(7) Otherwise the call is not linked.

Note that (1) takes precedence over (2), i.e., if a function appears on nolinkfns, the call to it is *not* linked, even if it is one of the functions in the block, i.e., the call will go outside of the block.

Nolinkfns is initialized to (HELP ERRORX ERRORSET EVALV FAULTEVAL INTERRUPT SEARCHPDL MAPDL BREAK1 EDITE EDITL). Linkfns is initialized to NIL. Thus if the user does not specify otherwise, all calls from a block compiled function (except for those to functions on nolinkfns) will be linked; all calls from standardly compiled functions will not be linked. However, when compiling system functions such as help, error, arglist, fntyp, breakl, et al, linkfns is set to T so that even though these functions are not block compiled, all of their calls will be linked.

If a function is not defined at link time, i.e., when an attempt is made to link to it, a message is printed. When the function is later defined, the link can be completed by relinking the calling function using relink described below. Otherwise, if a function is run which attempts a linked call that was not completed, interrupt is called as though an UNDEFINED FUNCTION CALL error had occurred, which in a sense it had. If the function is now defined, i.e., it was defined at some point after the attempt was made to link to it, interrupt will quietly perform the link and continue the call. Otherwise, it will print UNDEFINED FUNCTION CALL and proceed as described in Section 16.

Linked function calls are printed on the backtrace as ;fn; where fn is the name of the function. Note that this name does *not* actually appear on the stack, and that stkpos, retfrom, and the rest of the pushdown list functions (section 12) will *not* be able to find it. Functions which must be visible on the stack should not be linked to, i.e., include them on nolinkfns when compiling a function that would otherwise link its calls.

printstructure, calls, break on fn1-IN-fn2 and advise fn1-IN-fn2 all work correctly for linked function calls, e.g., break((FOO IN FIE)), where FOO is called from FIE via a linked function call.

Relinking

The function relink is available for relinking a compiled function, i.e., updating all of its linked calls so that they use the definition extant at the time of the relink operation.

relink[fn]

fn is either WORLD, the name of a function, a list of functions, or an atom whose value is a list of functions. relink performs the corresponding relinking operations. relink[WORLD] is possible because laprd maintains on linkedfns a list of all user functions containing any linked calls. syslinkedfns is a list of all system functions that have any linked calls. relink[WORLD] performs both relink[linkedfns] and relink[syslinkedfns].

The value of relink is fn.

It is important to stress that linking takes place when a function is *defined*. Thus, if foo calls fie via a linked call, and a bug is found in fie, fixing fie is not sufficient, foo must be relinked. Similarly, if fool, foo2, and foo3 are defined (in that order) in the file cfoo, and each call the others via linked calls, when a new version of the file cfoo is loaded, fool will be linked to the *old* foo2 and foo3, since those definitions will be extant at the time it is read and defined. Similarly, foo2 will link to the new fool and *old* foo3. Only foo3 will link to the new fool and foo2. The user would have to perform relink[FOOFNS] following the load.

The Block Compiler

There are three user level functions for blockcompiling, blockcompile, bcompl, and brecompile, corresponding to compile, tcompl, and recompile. All of them ultimately call the same low level functions in the compiler, i.e., there is no 'blockcompiler' per se. Instead, when blockcompiling, a flag is set to enable special treatment for specvars, retfns, blkapplyfns, and for determining whether or not to link a function call. Note that all of the previous remarks on macros, globalvars, compiler messages, etc., all apply equally for block compiling. Using block declarations described below, the user can intermix in a single file functions compiled normally, functions compiled normally with linked calls, and block compiled functions.

blockcompile

blockcompile[blkname;blkfns;entries;flg]

blkfns is a list of the functions comprising the block, blkname is the name of the block, entries a list of entries to the block, e.g.,

```
←BLOCKCOMPILE(SUBPRBLOCK (SUBPAIR SUBLIS SUBPR) (SUBPAIR SUBLIS))
```

Each of the entries must also be on blkfns or an error is generated.

If entries is NIL, list[blkname] is used, e.g.,

```
←BLOCKCOMPILE(COUNT (COUNT COUNT1))
```

If blkfns is NIL, list[blkname] is used, e.g.,

```
←BLOCKCOMPILE(EQUAL)
```

blockcompile asks the standard compiler questions and then begins compiling. As with compile, if the compiled code is being written to a file, the file is closed unless flg=T. The value of blockcompile is a list of the entries, or if entries=NIL, the value is blkname.

The output of a call to blockcompile is one function definition for blkname plus definitions for each of the functions on entries if any. These entry functions are very short functions which immediately call blkname.

Block Declarations

Since block compiling a file frequently involves giving the compiler a lot of information about the nature and structure of the compilation, e.g., block functions, entries, specvars, linking, et al, we have implemented a special prettydef command to facilitate this communication. The user includes in the third argument to prettydef a command of the form (BLOCKS block_1 ... block_2 ... block_n) where each block_i is a block declaration. bcompl and brecompile described below are sensitive to these declarations and take the appropriate action. If the user follows the convention of setting fileBLOCKS to a list of his block declarations, and then uses (BLOCKS * fileBLOCKS) in the third argument to prettydef, he will be able to use the more useful options of brecompile and cleanup.

The form of a block declaration is

(blkname blkfn_1 ... blkfn_m (var_1.value) ... (var_n.value))

blkfn_1 ... blkfn_m are the functions in the block and correspond to blkfns in the call to blockcompile. The (var.value) expressions indicate the settings for variables affecting the compilation.

As an example, the value of editblocks is shown below. It consists of three block declarations, editblock, editfindblock, and edit4e.

```
(RPAQQ EDITBLOCKS
       ((EDITBLOCK EDITL EDITL1 EDITEXIT UNDOEDITL EDITCOM EDITCOMA
                   EDITCOML EDITMAC EDITCOMS EDIT!UNDO UNDOEDITCOM
                   EDITSMASH EDITNCONC EDIT1F EDIT2F EDITNTH BPNT
                   BPNTØ BPNT1 RI RO LI LO BI BO EDITDEFAULT ## EDUP
                   EDIT* EDOR EDRPT EDLOC EDLOCL EDIT: EDITMBD
                   EDITXTR EDITELT EDITCONT EDITSW EDITMV EDITTO
                   EDITBELOW EDITRAN TAILP EDITSAVE EDITH
                   (ENTRIES EDITL ## UNDOEDITL)
                   (SPECVARS L COM #1 #2 #3 UNFIND LASTAIL MARKLST
                             LISPXBUFS)
                   (RETFNS EDITE EDITL)
                   (GLOBALVARS EDITCOMSA EDITCOMSL EDITOPS
                               HISTORYCOMS EDITRACEFLG)
                   (BLKAPPLYFNS RI RO LI LO BI BO EDIT: EDITMBD
                                EDITMV EDITXTR)
                   (BLKLIBRARY LENGTH NTH))
        (EDITFINDBLOCK EDIT4E EDITQF EDIT4F EDITFPAT EDIT4F1
                       EDITFINDP EDITBF EDITBF1 ESUBST
                       (ENTRIES EDITQF EDIT4F EDITFPAT EDITFINDP
                                EDITBF ESUBST))
        (EDIT4E)))
```

Whenever bcompl or brecompile encounter a block declaration* they rebind retfns, specvars, globalvars, blklibrary, nolinkfns, and linkfns to their top level value, bind blkapplyfns and entries to NIL, and bind blkname to the first element of the declaration. They then scan the rest of the declaration, gathering up all atoms, and setting car of each nonatomic element to cdr of the expression if atomic, e.g., (LINKFNS . T), or else to union of cdr of the expression with the current (rebound) value, e.g., (GLOBALVARS EDITCOMSA EDITCOMSL). When the declaration is exhausted, the block compiler is called and given blkname, the list of block functions, and entries.

* The BLOCKS command outputs a DECLARE expression, which is noticed by bcompl and brecompile.

Note that since all compiler variables are rebound for each block declaration, the declaration only has to set those variables it wants *changed*. Furthermore, setting a variable in one declaration has no effect on the variable's value for another declaration.

After finishing all blocks, bcompl and brecompile treat any functions in the file that did not appear in a block declaration in the same way as do tcompl and recompile. If the user wishes a function compiled separately as well as in a block, or if he wishes to compile some functions (not block compile), with some compiler variables changed, he can use a special pseudo-block declaration of the form

(NIL fn_1 ... fn_m (var_1 . value) ... (var_n . value))

which means compile fn_1 ... fn_m after first setting var_1 ... var_n as described above.

For example,

```
(NIL CGETD FNTYP ARGLIST NARGS NCONC1 GENSYM (LINKFNS . T))
```

appearing as a 'block declaration' will cause the six indicated functions to be compiled while linkfns=T so that all of their calls will be linked (except for those functions on nolinkfns).

bcompl

bcompl[files;cfile]

files is a list of prettydefed files. (If atomic, list[files] is used.) bcompl differs from tcompl in that it compiles all of the files at once, instead of one at a time. This is to permit one block to contain functions in several files.* Output is to cfile if given, otherwise to a file whose name is car[files] suffixed with COM** e.g.,

bcompl[(EDIT WEDIT)]

produces one file, EDIT.COM.

bcompl asks the standard compiler questions, except for OUTPUT FILE: then reads in all of the files, adds all function in defineqs to nlama, nlaml, and then processes the block declarations as described above. Finally, it compiles any functions not mentioned in one of the declarations and writes out all other expressions, e.g., RPAQQ'S.

---

* Thus if you have several files to be bcompled *separately*, you must make several calls to bcompl.

**See footnote, p. 18.10.

The value of <u>bcompl</u> is the output file.

Note that it is permissible to <u>tcompl</u> files set up for <u>bcompl</u>; the block declarations will simply have no effect. Similarly, you can <u>bcompl</u> a file that does not contain any block declarations and the result will be the same as having <u>tcompl</u>ed it.

Brecompile

The purpose of brecompile is to allow the user to update a compiled file without requiring an entire bcompl. As with recompile, the usual way to call brecompile involves specifying just its first argument, the symbolic file(s), as in the sequence of loading file(s) to PROP, editing selected definitions, makefiling, and then calling brecompile. In this case, brecompile recompiles all exprs and works from in-core definitions.* Note that this assumes that each symbolic file was produced by makefile, i.e., the arguments to prettydef were fileFNS, file, and fileVARS, since brecompile uses fileFNS and fileVARS to drive its operation. The rest of the discussion below is for various nonstandard usages.

brecompile[files;cfile;fns;coreflg] files is a list of symbolic files (if atomic, list[files] is used). cfile is the compiled file corresponding to bcompl[files] or a previous brecompile, i.e., it contains compiled definitions that may be copied.

fns is a list of those functions to be recompiled, i.e., they have been changed (or defined for the first time) since cfile was made. If fns=T, all functions defined as exprs (after unbreaking and unadvising) are recompiled.

* Note that if any of the functions in a block are recompiled, the entire block is recompiled.

brecompile asks the standard compiler questions except for OUTPUT FILE: As with bcompl, output automatically goes to file.COM, where file is the first file in files.

If coreflg=NIL, brecompile proceeds to read in each file, collecting all definitions while making the appropriate additions to nlama and nlaml, and collecting all block declarations, and other expressions which will later be copied to the output file.

If coreflg=T, the value of fileBLOCKS is used for the block declarations, where file is the first file in files, and fileFNS and fileVARS are used as a representation of what actually appears on each file in files.* The only access to the files is to obtain the date for each file so that it can be written on the output file.

brecompile next processes each block declaration. If no functions in the block have been changed, the block is copied from cfile as with recompile.

* See footnote, p. 18.10.

Otherwise, the entire block is recompiled. For pseudo-block declarations of the form (NIL fnl ...), all variable assignments are made, but only those functions so indicated by fns are recompiled.

As with recompile, the order in which functions appear on the file must not be changed unless all of the functions that are moved are also recompiled.

After completing the block declarations, brecompile processes all functions not appearing in a declaration, recompiling only those dictated by fns, and copying the compiled definitions of the remaining from cfile.

Finally, brecompile writes the portion of file.COM corresponding to the nonDEFINEQ expressions. If coreflg=NIL, brecompile simply writes out those expressions which it had previously collected. Otherwise, it uses fileVARS to determine what is on each file and writes the corresponding expressions on to the output file.

The value of brecompile is the output file.

If cfile=NIL, file.COM is used,* coreflg is set to T, and if fns is NIL, it is set to T. This is the standard usage described earlier.

Note brecompile should *not* be used if some of the functions in a file were written via a FNS command, instead of via the first argument to prettydef.

* See footnote, p. 18.12.

Compiler Structure

The compiler has two principal passes. The first compiles its input into a macro assembly language called LAP. The second pass expands the LAP code, producing (numerical) machine language instructions. The output of the second pass is written on a file and/or stored in binary program space.

Input to the compiler is usually a standard LISP S-expression function definition. However, machine language coding can be included within a function by the use of one or more assemble forms. In other words, assemble allows the user to write portions of a function in LAP. Note that assemble is only a compiler directive; it has no independent definition. Therefore, functions which use assemble must be compiled in order to run.

Assemble

The format of assemble is similar to that of PROG.

(ASSEMBLE V S1 S2 . . . SN)

V is a list of variables to be bound during the first pass of the compilation, *not* during the running of the object code. The assemble statements S1 ... SN are compiled sequentially, each resulting in one or more instructions of object code. When run, the value of the assemble 'form' is the contents of AC1 at the end of the execution of the assemble instructions. Note that assemble may appear anywhere in a LISP function. For example, one may write

```
(IGREATERP (IQUOTIENT (LOC (ASSEMBLE NIL
                                     (MOVEI 1 , -5)
                                     (JSYS 13)))
                       1ØØØ)
           4)))
```

to test if job runtime exceeds 4 seconds.

Assemble Statements

If an assemble statement is an atom, it is treated as a label identifying the location of the next statement that will be assembled.[†] Such labels defined in an assemble form are local to that assemble form.

If an assemble statement is not an atom, car of the statement must be an atom and one of the following: (1) a number; (2) a LAP op-def (i.e. has a property value OPD); (3) an assembler macro (i.e. has a property value AMAC); or (4) one of the special assemble instructions given below, e.g. C, CQ, etc. Anything else will cause the error message OPCODE? - ASSEMBLE.

The types of assemble statements are described here in the order of priority used in the assemble processor; that is, if an atom has both properties OPD and AMAC, the OPD will be used. Similarly a special assemble instruction may be redefined via an AMAC. The following descriptions are of the first pass processing of assemble statements. The second pass processing is described in the section on LAP, p. 18.42.

(1) numbers - If car of an assemble statement is a number, the statement is not processed in the first pass. (See page 18.42.)

(2) LAP op-defs - The property OPD is used for two different types of op-defs: PDP-10 machine instructions, and LAP macros. If the OPD definition (i.e. the property value) is a number, the op-def is a machine instruction. When a machine instruction, e.g. HRRZ, appears as car of an assemble statement, the statement is not processed during the first pass but is

[†]A label can be the last thing in an assemble form, in which case it labels the location of the first instruction *after* the assemble form.

passed to LAP. The forms and processing of machine instructions by LAP are described on page 18.42.

If the OPD definition is not a number, then the op-def is a LAP macro. When a LAP macro is encountered in an assemble statement, its arguments are evaluated and processing of the statement with evaluated arguments is left for the second pass and LAP. For example, STT is a LAP macro, and (STT (PSTEP)) in assemble code results in (STT n) in the LAP code, where n is the value of (PSTEP).

The form and processing of LAP macros are described on page 18.45.

(3) assemble macros - If <u>car</u> of an assemble statement has a property AMAC, the statement is an assemble macro call. There are two types of assemble macros: lambda and substitution. If <u>car</u> of the macro definition is the atom LAMBDA, the definition will be *applied* to the arguments of the call and the resulting list of statements will be assembled. For example, <u>repeat</u> is a LAMBDA macro with two arguments, <u>n</u> and <u>m</u>, which expands into <u>n</u> occurrences of <u>m</u>, e.g. (REPEAT 3 (CAR1)) expands to ((CAR1) (CAR1) (CAR1)). The definition (i.e. value of property AMAC) for <u>repeat</u> is:

```
(LAMBDA (N M)
  (PROG (YY)
    A   (COND
          ((ILESSP N 1)
            (RETURN (CAR YY)))
          (T (SETQ YY (TCONC YY M))
             (SETQ N (SUB1 N))
             (GO A))))))
```

If <u>car</u> of the macro definition is not the atom LAMBDA, it must be a list of dummy symbols. The arguments of the macro call will be substituted for corresponding appearances of the dummy symbols in <u>cdr</u> of the definition and the resulting list of statements will be assembled.[†] For example, <u>ubox</u> is a substitution macro which takes one argument which is a number, and expands into instructions to compile the unboxed value of this number and put the result on the number stack.

The definition of UBOX is:

```
        ((E)
       (CQ (VAG E))
       (PUSH NP , 1))
```

Thus (UBOX (ADD 1 X))

expands to

```
       ((CQ (VAG (ADD1 X)))
        (PUSH NP , 1))
```

(4) special assemble statements

(CQ s_1 s_2 ...)	CQ (<u>c</u>ompile <u>q</u>uote) takes any number of arguments which are assumed to be regular S-expressions and are compiled in the normal way. E.g. (CQ (COND ((NULL Y) (SETQ Y 1))) (SETQ X (IPLUS Y Z)))

To avoid confusion, it is best to have as much of a function as possible compiled in the normal way, i.e. to load the value of <u>x</u> to AC1, (CQ X) is preferred to (E (LDCOMP (QUOTE X) 1)).

† Note that assemble macros produce a list of statements to be assembled, whereas compiler macros produce a single expression. An assemble macro which *computes* a list of statements begins with LAMBDA and may be *either* spread or no-spread. The analagous compiler macro begins with an atom, (i.e. is always no-spread) and the LAMBDA is understood.

(C s_1 s_2 ...)	C (compile) takes any number of arguments which are first evaluated, then compiled in the usual way. Both C and CQ permit the inclusion of regular compilation within an assemble form.
(E e_1 e_2 ...)	E (evaluate) takes any number of arguments which are evaluated in sequence. For example, (E (LDCOMP (QUOTE X) 3)) calls a function which produces code to load the value of x to AC3.
(SETQ var)	Compiles code to set the variable var to the contents of AC1.
(FASTCALL fn)	Compiles code to call fn. Fn must be one of the SUBR's that expects its arguments in the accumulators, and not on the push-down stack. Currently, these are cons, and the boxing and unboxing routines.† Example: (CQ X) (E (LDCOMP (QUOTE Y) 2)) (FASTCALL CONS) and cons[x,y] will be in AC1.
(* ...)	* is used to indicate a comment; the statement is ignored.

† list may also be called with fastcall by placing its arguments on the pushdown stack, and the *number* of arguments in AC1.

COREVALS

There are several locations in the basic machine code of LISP which may be referenced from compiled code. The current value of each location is stored on the property list under the property COREVAL.[†] Since these locations may change in different versions of LISP, they are written symbolically on compiled code files, i.e. the name of the corresponding COREVAL is written, not its value. Some of the COREVALS used frequently in assemble are:

CONS	entry to function CONS
LIST	entry to function LIST
KT	contains (pointer to) atom T
KNIL	contains (pointer to) atom NIL
MKN	routine to box an integer
MKFN	routine to box floating number
IUNBOX	routine to unbox an integer
FUNBOX	routine to unbox floating number

The index registers used for the push-down stack pointers are also included as COREVALS. These are not expected to change, and are not stored symbolically on compiled code files; however, they should be referenced symbolically in assemble code. They are:

PP	parameter stack
CP	control stack
NP	number stack

[†]The value of corevals is a list of all atoms with COREVAL properties.

LAP

LAP (for LISP assembly processor) expands the output of the first pass of compilation to produce numerical machine instructions.

LAP Statements

If a LAP statement is an atom, it is treated as a label identifying the location of the next statement to be processed. If a LAP statement is not an atom, car of it must be an atom and one of the following: (1) a number; (2) a machine instruction; or (3) a LAP macro.

(1) numbers - If car of a LAP statement is a number, a location containing the number is produced in the object code.

```
    e.g.     (ADD 1 , A (1))
              .
              .
              .
       A     (1)
             (4)
             (9)
```

Statements of this type are processed like machine instructions, with the initial number serving as a 36-bit op-code.

(2) Machine Instructions - If car of a LAP statement has a numeric value for the property OPD,† the statement is a machine instruction. The general form of a machine instruction is:

```
(opcode ac , @ address (index))
```

Opcode is any PDP-10 instruction mnemonic or LISP UUO.††

†The value is an 18 bit quantity (rather than 9), since some UUO's also use the AC field of the instruction.

††The TENEX JSYS's are not defined, that is, one must write (JSYS 107) instead of (KFORK).

Ac, the accumulator field, is optional. However, if present, it *must* be followed by a comma. Ac is either a number or an atom with a COREVAL property. The low order 4 bits of the number or COREVAL are OR'd to the AC field of the instruction.

@ may be used anywhere in the instruction to specify indirect addressing (bit 13 set in the instruction) e.g. (HRRZ 1 , @ ' V).

Address is the address field which may be any of the following:

= constant	Reference to an unboxed constant. A location containing the unboxed constant will be created in a region at the end of the function, and the address of the location containing the constant is placed in the address field of the current instruction. The constant may be a number e.g. (CAME 1 , = 3596); an atom with a property COREVAL (in which case the constant is the value of the property, at LOAD time); any other atom which is treated as a label (the constant is then the address of the labeled location) e.g. (MOVE 1 , = TABLE) is equivalent to (MOVEI 1 , TABLE); or an expression whose value is a number.

' pointer	The address is a reference to a LISP pointer, e.g. a list, number, string, etc. A location containing the pointer is assembled at the end of the function, and the current instruction will have the address of this location. E.g. (HRRZ 1 , ' "IS NOT DEFINED") (HRRZ 1 , ' (NOT FOUND))
*****	Specifies the current location in the compiled function; e.g. (JRST * 2) has the same effect as (SKIPA).
literal atom	If the atom has a property COREVAL, it is a reference to a system location, e.g. (SKIPA 1 , KNIL), and the address used is the value of the <u>coreval</u>. Otherwise the atom is a label referencing a location in the LAP code, e.g. (JRST A).
number	The number is the address; e.g. (MOVSI 1 , 400000Q) (HLRZ 2 , 1 (1))
list	The form is evaluated, and its value is the address.

Anything else in the address field **causes** an error message, e.g. (SKIPA 1 , KNILL) - LAPERROR. A number may follow the address field and will be added to it, e.g. (JRST A 2).

Index is denoted by a *list* following the address field, i.e. the address field *must* be present if an index field is to be used. The index (car of the list) must be either a number, or an atom with a property COREVAL, e.g. (HRRZ 1 , 0 (1)) or (ANDM 1 , -1 (NP))

(3) **LAP** macros - If car of a LAP statement is the name of a LAP macro, i.e. has the property OPD, the statement is a macro call. The arguments of the call follow the macro name: e.g. (LQ2 FIE 3).

Lap macro calls comprise most of the output of the first pass of the compiler, and may also be used in assemble. The definitions of these macros are stored on the property list under the property OPD, and like assembler macros, may be either lambda or substitution macros. In the first case, the macro definition is applied to the arguments of the call;[†] in the second case, the arguments of the call are substituted for occurrences of the dummy symbols in the definition. In both cases, the resulting list of statements is again processed, with macro expansion continuing till the level of machine instructions is reached.

Some examples of LAP macros are:

[†]The arguments were already evaluated in the first pass, see top of page 18.38.

```
(DEFLIST (QUOTE ((SVN ((N P)                          (* STORE VARIABLE NAME)
                       (MOVE 1 , ' N)
                       (HRLM 1 , P (PP))))
                 (SVB ((N P)                          (* STORE VARIABLE NAME
                                                      AND VALUE)
                       (HRL 1 , ' N)
                       (MOVEM 1 , P (PP))))
                 (STT ((N)                            (* STORE TEMPORARY)
                       (HRRZM 1 , N (PP))))
                 (LDT ((N)                            (* LOAD TEMPORARY)
                       (HRRZ 1 , N (PP))))
                 (LQ ((X)                             (* LOAD QUOTE)
                      (HRRZ 1 , ' X)))
                 (LQ2 ((X AC)                         (* LOAD QUOTE TO AC)
                       (HRRZ AC , ' X)))
                 (LQI ((X)                            (* LOAD QUOTE IMMEDIATE
                                                      -
                                                      FOR SMALL NUMBERS)
                       (HRRZI 1 , ASZ X)))
                 (LDV ((A)                            (* LOAD LOCAL VARIABLE)
                       (HRRZ 1 , (VREF A))))
                 (STV ((A)                            (* SET LOCAL VARIABLE)
                       (HRRM 1 , (VREF A))))
                 (LDV2 ((A AC)                        (* LOAD LOCAL VARIABLE
                                                      TO AC)
                        (HRRZ AC , (VREF A))))
                 (LDF ((A)                            (* LOAD FREE VARIABLE)
                       (HRRZ 1 , (FREF A))))
                 (CAR1 (NIL                           (* CAR OF AC 1)
                            (HRRZ 1 , Ø (1))))
                 (CDR1 (NIL                           (* CDR AC 1)
                            (HLRZ 1 , Ø (1))))
                 (CARQ ((V)                           (* CAR QUOTE)
                        (HRRZ 1 , @ ' V)))
                 (CAR2 ((AC)                          (* CAR AC)
                        (HRRZ AC , Ø (AC))))
                 (RPQ ((V)                            (* RPLACA QUOTE)
                       (HRRM 1 , @ ' V)))
                 (CLL ((NAM N SP)                     (* CALL FN WITH N ARGS
                                                      GIVEN)
                       (CLLS N SP)
                       (MOVE 2 , ' NAM)
                       (XCT FNCALL)))
                 (STE ((TY)                           (* SKIP IF TYPE EQUAL)
                       (PSTE1 TY)))
                 (STN ((TY)                           (* SKIP IF TYPE NOT
                                                      EQUAL)
                       (PSTN1 TY)))
                 (RET (NIL                            (* RETURN FROM FUNCTION)
                           (POPJ CP ,)))
                 (POPNN ((N)                          (* POP N ENTRIES FROM
                                                      NUMBER STACK)
                         (SUB NP , BHC N)))))
         (QUOTE OPD))
```

Using Assemble

In order to use assemble, it is helpful to know the following things about how compiled code is run. All variable bindings and temporary values are stored on the parameter push-down stack. When a compiled function is entered, the parameter push-down list contains, in ascending order of address:

1. bindings of arguments to the function, where each binding occupies one word on the stack with the variable name in the left half and the value in the right half.
2. pointers to the most recent bindings of free variables used in the function.

The parameter push-down list pointer, index register PP, points to the last free variable pointer on the stack.

Temporary values, PROG and LAMBDA bindings, and the arguments to functions about to be called, are stored following the free variable pointers. However, the pointer PP is not changed until the lower function is actually called. **The compiler uses the** value of the variable SP to keep track of the *number* of positions in use beyond the current value of PP, so that it knows where to store the next temporary value. The function PSTEP adds 1 to SP, and PSTEPN(N) adds N to SP (N can be positive or negative).

The parameter stack should only be used for storing pointers. In addition, anything in the left half of a word on the stack is assumed to be a variable name (see p. 12.2). To store unboxed numbers, use the number stack, NP. Numbers may be PUSH'ed and POP'ed on the number stack.

Miscellaneous

The value of a function is always returned in AC1. Therefore, the pseudo-function, ac, is available for obtaining the current contents of AC1. For example (CQ (FOO (AC))) compiles a call to FOO with the current contents of AC1 as argument, and is equivalent to

```
(STT (PSTEP))
(CLL (QUOTE FOO) 1 SP)
(E (PSTEPN -1))
```

In using ac be sure that it appears as the first argument to be evaluated in the expression. For example

```
(CQ (IPLUS (LOC (AC)) 2))
```

⁙ ⁙ ⁙

There are several ways to reference the values of variables in assemble code. For example:

(CQ X)	puts value of X in AC1
(E (LDCOMP (QUOTE X) 3)	puts value of X in AC3
(SETQ X)	sets X to contents of AC1
(HRRM 2 , (VREF X))	sets X to contents of AC2

Vref can be used in the address field to reference a local or free variable. However a free variable *must* be referenced somewhere in the function by something that is processed during the first pass of compilation, because the compiler must know *before* the second pass how many free variables are used in the function. Varcomp is a function that 'notices' a free variable, i.e. including (E (VARCOMP (QUOTE var))) will solve the above problem, and not generate any instructions. Varcomp should also be used whenever vref is used and there is a possibility that the variable is free and is not referenced (i.e. has not been noticed) elsewhere in the function.

⁙ ⁙ ⁙

To box and unbox numbers:

(CQ (LOC (AC)))	box contents of AC1
(FASTCALL MKN)	box contents of AC1
(FASTCALL MKFN)	floating box contents of AC1
(CQ (VAG X))	unboxed value of X to AC1
(FASTCALL IUNBOX)	unboxed contents of AC1
(FASTCALL FUNBOX)	floating unbox of AC1

To call a function directly, the arguments must be put on the parameter stack, and SP must be updated, and then the function called: e.g.

```
        (CQ (CAR X)
        (STT (PSTEP))              (* stack first argument)
        (HRRZ 1 , ' 3.14)
        (STT (PSTEP))              (* stack second argument)
        (CLL (QUOTE FUM) 2 SP)     (* call FUM with 2 arguments)
        (E (PSTEPN -2)             (* adjust stack count)
```

is equivalent to

```
        (CQ (FUM (CAR X) 3.14))
```

SECTION XIX

ADVISING

Contents

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, and breakdown, 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 to set sysdate to the time and date of creation by advise[SYSOUT;(SETQ SYSDATE (DATE))]

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.

For example, suppose the user wanted time (section 21) to print the results of his measurements to the file FOO instead of the teletype. He could accomplish this by

```
    ADVISE (((PRIN1 PRINT SPACES) IN TIME) (SETQQ U FOO))
```

Note that advising prin1, print, or spaces directly would have affected all calls to these very frequently used functions, whereas advising ((PRIN1 PRINT SPACES) IN TIME) affects just those calls to prin1, print, and spaces from time.

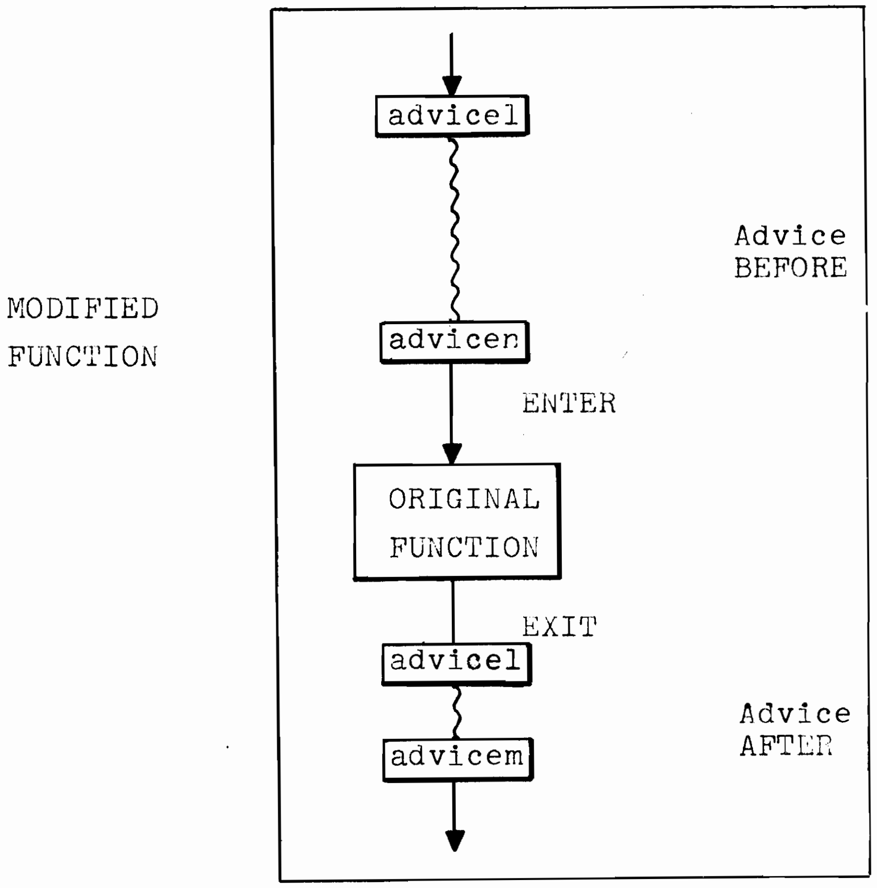
Advice can also be specified to operate after a function has been evaluated. The value of the body of the original function can be obtained from the variable !value, as with break1. For example, suppose the user wanted to perform some computation following each sysin, e.g. check whether his files were up to date. He could then

```
        ADVISE(SYSOUT AFTER (COND ((EQ !VALUE T) --)))*
```

*After the sysin, the system will be as it was when the sysout was performed, hence the advice must be to sysout, not sysin. See Section 14 for complete discussion of sysout/sysin.

Implementation of Advising

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
                   advice1
                     .
                     .                               ADVICE
                     .                               BEFORE
                   advicen
                    (RETURN form)))
              advice1
                .
                .                                    ADVICE
                .                                    AFTER
              advicem
               (RETURN !VALUE)))
```

where form is equivalent to the original definition.*

Note that the structure of a function modified by advise allows a piece of advice to bypass the original definition by using the function RETURN. For example, if (COND ((ATOM X) (RETURN Y))) were one of the pieces of advice BEFORE a function, and this function was entered with x atomic, y would be returned as the value of the inner PROG. !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.

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.

*If fn was originally an expr, form is the body of the definition, otherwise a form using a gensym which is defined with the original definition.

Advise Functions

Advise

Advise is a function of four arguments: fn, when, where, and what. fn is the function to be modified by advising, what is the modification, or piece of advice. when is either BEFORE or AFTER and indicates whether the advice is to operate BEFORE or AFTER the body of the function definition is evaluated. where specifies exactly where in the list of advice the new advice is to be placed, e.g., FIRST, or (BEFORE PRINT) meaning before the advice containing print, or (AFTER 3) meaning after the third piece of advice, or even (: TTY:). If where is specified, advise first checks to see if it is one of LAST, BOTTOM, END, FIRST, or TOP. Otherwise, it constructs an appropriate edit command and calls the editor to insert the advice at the appropriate location.

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: [fn;what], or a function of three arguments: [fn;when;what], or a function of four arguments: [fn;when;where;what]. Note that the advice is always the *last* argument. If when=NIL, BEFORE is used. If where=NIL, LAST is used.

advise[fn;when;where;what]

fn 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.

If fn is of the form (fn1 IN fn2), fn1 is changed to fn1-IN-fn2 throughout fn2, as with break, and then fn1-IN-fn2 is used in place of fn.*

If fn is broken, it is unbroken before advising.

If fn is not defined, an error is generated.

* If fn1 and/or fn2 are lists, they are distributed, see example p. 19.2.

If fn is being advised for the first time, i.e. if getp[name,ADVISED]=NIL, a gensym is generated and stored on the property list of fn under the property ADVISED, and the gensym is defined with the original definition of fn. An appropriate S-expression definition is then created for fn. Finally, fn is added to the (front of) advisedfns.*

If fn has been advised before, it is moved to the front of advisedfns.*

The advice is inserted in fn's definition either BEFORE or AFTER the original body function depending on when**. Within that context, its position is determined by where. If where=LAST, BOTTOM, END, or NIL, the advice is added following all other advice, if any. If where=FIRST or TOP, the advice is inserted as the first piece of advice. Otherwise, where is treated as a command for the editor, a la breakin, e.g.
(BEFORE 3),
(AFTER PRINT) .

* So that unadvise[T] always unadvises the last function advised. See p. 19.8.

** A special case is when=BIND. Here the advice is treated as a list of prog variables to be bound. The variables are nconced to the prog variable list containing !value. See p. 19.4.

Finally list[when;where;what] is added (by addprop) to the value of property ADVICE on the property list fn.* Note that this property value is a list of the advice in order of calls to advise, not necessarily in order of appearance of the advice in the definition of fn.

The value of advise is fn.

If fn is non-atomic, every function in fn is advised with the same values (but copies) for when, where, and what. In this case, the value of advise is a list of individual functions.

Note: advised functions can be broken. (However if a function is broken at the time it is advised, it is first unbroken.) Similarly, advised functions can be edited, including their advice. unadvise will still restore the function to its unadvised state, but any changes to the body of the definition will survive. Since the advice stored on the property list is the same list structure as the advice inserted in the function, editing of advice can be performed on either the function's definition or its property list.

---

* So that a record of all the changes is available for subsequent use in readvising, see p. 19.8, 19.9.

unadvise[x]

is a no-spread NLAMBDA a la unbreak. It takes an indefinite number of functions and restores them to their original unadvised state, including removing the properties added by advise.* unadvise saves on the list advinfolst enough information to allow restoring a function to its advised state using readvise. advinfolst and readvise thus correspond to brkinfolst and rebreak.

unadvise[] unadvises all functions on advisedfns.** It first sets advinfolst to NIL.

unadvise[T] unadvises the first function of advisedfns, i.e., the most recently advised function.

---

* Except if a function also contains the property READVICE (see readvise below), unadvise moves the current value of the property ADVICE to READVICE.

**In reverse order so that the most recently advised function is unadvised last.

readvise[x]

is a no-spread NLAMBDA a la rebreak. for restoring a function to its advised state without having to respecify all the advise information. For each function on x, readvise retrieves the advise information either from the property READVICE for that function, or from advinfolst, and performs the corresponding advise operation(s). In addition, it stores this information on the property READVICE if not already there. If no information is found for a particular function, value is (fn - NO ADVICE SAVED).

readvise[] readvises everything on advinfolst.

readvise[T] readvises just the first function on advinfolst, i.e., the function most recently unadvised.

The difference between advise, unadvise, and readvise versus break, unbreak, and rebreak, is that if a function is not rebroken between successive unbreak[]'s, its break information is forgotten. However, once readvised, a function's advice is permanently saved on its property list (under READVICE); subsequent calls to unadvise will not remove it. In fact, calls to unadvise update the property READVICE with the current value of ADVICE, so that the sequence readvise, advise, unadvise causes the augmented advice to become permanent. Note that the sequence readvise, advise, readvise removes the 'intermediate advice' by restoring the function to its earlier state.

advisedump[x;flg] Used by prettydef when given a command of the form (ADVISE --) or (ADVICE --). See p. 14.30. flg=T corresponds to (ADVISE --), i.e. advisedump writes both a deflist and a readvise. flg=NIL corresponds to (ADVICE --), i.e. only the deflist is written. In either case, advisedump copies the advise information to the property READVICE, thereby making it 'permanent' as described above.

SECTION XX

PRINTSTRUCTURE

Contents

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 illustrate this in more detail, we use the printstructure program itself as an example.

```
PRINTSTRUCTURE  PRGETD
                PROGSTRUC PRGETD
                          PRGSTRC   NOTFN     PRGETD
                                    PROGSTRUC
                                    PRGSTRC1  PRNCONC
                                              PRGSTRC1
                                              PRGSTRC
                                    PRNCONC
                                    PRGSTRC
                          CALLS1    MAKELIST
                                    NOTFN
                                    CALLS2    CALLS1
                                              PRGETD
                TREEPRINT TREEPRINT1
                          TREEPRINT
                VARPRINT  VARPRINT1 TREEPRINT1
                          VARPRINT2 ALLCALLS  ALLCALLS1 ALLCALLS1
                          TREEPRINT1

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

PRINTSTRUCTURE  [X,FILE; DONELST,N,TREELST,TREEFNS,L,TEM,X,Y,Z,FN,TREE;
PRDEPTH,LAST-PRINTSTRUCTURE]
   CALLED BY:

PRGETD    [X,FLG; ; ]
   CALLED BY:  PRINTSTRUCTURE,PROGSTRUC,NOTFN,CALLS2

PROGSTRUC [FN,DEF; N,Y,Z,CALLSFLG,VARSFLG,VARS1,VARS2,D,X; N,DONELST]
   CALLED BY:  PRINTSTRUCTURE,PRGSTRC

PRGSTRC   [X,HEAD,FLG; Y,TEM,X; VARSFLG,D,NOFNS,CALLSFLG,N,DONELST,
TREEFNS,NOTRACEFNS,FN,VARS1,QUOTEFNS]
   CALLED BY:  PROGSTRUC,PRGSTRC1,PRGSTRC

NOTFN     [FN; DEF; NOFNS,YESFNS,FIRSTLOC,LASTLOC]
   CALLED BY:  PRGSTRC,CALLS1

PRGSTRC1  [L,HEAD,FLG; A,B; VARS1,VARS2]
   CALLED BY:  PRGSTRC,PRGSTRC1

PRNCONC   [X,Y; ; CALLSFLG]
   CALLED BY:  PRGSTRC1,PRGSTRC

CALLS1    [ADR,GENFLG,D; LIT,END,V1,V2,LEFT,OPD,X,X; VARS1,VARS2,VARSFLG
]
   CALLED BY:  PROGSTRUC,CALLS2

MAKELIST  [N,ADR; L; ]
   CALLED BY:  CALLS1
```

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 prgetd, progstruc, treeprint, and varprint. progstruc in turn calls prgetd, prgstrc and calls1. prgstrc calls notfn, progstruc, prgstrc1, prnconc, and itself. prgstrc1 calls prnconc, 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, for each function, information about the variables it uses and a list of the functions that call it. For example, printstructure is a function of two arguments, x and file. It binds eleven variables internally: donelst, n, ... tree*, and uses prdepth and last-printstructure as free variables. It is not called by any of the functions in the tree. prgetd is a function of two arguments, x and flg, binds no variables internally, uses no free variables, and is called by printstructure, progstruc, notfn and calls2.

printstructure calls many other low-level functions such as getd, car, list, nconc, etc. in addition to the four functions appearing in the above output. The reason these do not appear in the output is that they were defined "uninteresting" by the user for the purposes of his analysis. Two functions, firstfn and lastfn, and two variables, yesfns and nofns are used for this purpose. Any function that appears on the list nofns is not of interest, any function appearing on yesfns is of interest.

* Variables are bound internally by either progs or open lambda-expressions.

yesfns=T effectively puts all functions on yesfns.* As for functions appearing on neither nofns or yesfns, all interpreted functions are deemed interesting, but only those compiled functions whose code lies in that portion of bpspace between the two limits established by firstfn and lastfn. For example, the above analysis was performed following firstfn[printstructure], lastfn[allcallsl].

Three other variables, notracefns, quotefns, and prdepth also affect the action of printstructure. Functions that appear on the list notracefns will appear in the tree, assuming they are "interesting" functions as defined above, but their definitions will not be analyzed.

Functions that appear on quotefns are analyzed, assuming they are "interesting," but when they appear as car of a form, the rest of the form, i.e., the arguments, is not analyzed. For example, if the function prinq were defined as (NLAMBDA (X) (MAPC X (FUNCTION PRIN1))) and included on quotefns, and the form (PRINQ (NOW IS THE TIME)) appeared in a function being analyzed, prinq *would* appear in the tree and be analyzed but the 'form' (NOW IS THE TIME) would be skipped. The initial setting of quotefns is NLAMBDAS, which effectively includes all NLAMBDAS (functions with argtype 1 or 3) on quotefns, except for those functions which printstructure knows require evaluation, e.g., ersetq, nlsetq, or, and, etc. The arguments to these functions are always analyzed.

*The decision as to whether or not a function is interesting is handled by the function notfn which returns T if the function is *not* interesting. It is notfn that checks nofns and yesfns. Accordingly, notfn can be modified or advised to perform arbitrary computations, e.g., any function shorter than 100 instructions in length is 'uninteresting', etc. The following functions are rejected by a special check in notfn: cond, prog, go, assemble, progn, selectq, function, quote, *, or, and, not, null, eq, neq, setq, return, car, cdr, cadr, cddr, caddr, cdddr, atom, iplus, itimes, ilessp, igreaterp, cons, list, mapc, and map. Thus these functions will never be printed in the tree unless they are specifically included on yesfns, or yesfns is set to T.

Finally, prdepth is a cutoff depth for analysis. It is initially set to 7.

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 or open lambda expression) 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 since printstructure detects functions that are not defined, (i.e., atoms appearing as CAR of a form), printstructure is a useful tool for debugging.

```
←PP FOO

(FOO
  [LAMBDA (X)
    (COND
      ((CAR X) (FOO1 X))
      (T ((CONS X (CAR X])
FOO
←PRINTSTRUCTURE(FOO)

FOO          FOO1

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

FOO          [X; ; ]
   CALLED BY:

FOO1         IS NOT DEFINED.

P.P.E. IN FOO - ((CONS X (CAR X)))
```

* except there may be some confusion in analyzing compiled functions, if the name of a variable and a function are the same.

Other Options

printstructure is a function of two arguments, x and file. printstructure analyzes x, sets the free variable last-printstructure to the results of its analysis, prints the result (in the format shown earlier) to file (which is opened if necessary and closed afterwards), and returns x as its value. Thus if the user did not want to see any output, he could call printstructure with file=NIL:,* and then process the result himself by using last-printstructure.

x can be NIL, a list, a function, or an atom that evaluates to a list. If x is NIL, printstructure does not perform any analysis, but simply prints the result of the last analysis, i.e., that stored on last-printstructure. Thus the user can effectively redirect the output of printstructure to a disc file by aborting the printout, and then performing printstructure[NIL,file].

If x is a list, printstructure analyzes the first function on x, and then analyzes the second function, *unless it was already analyzed,* then the third, etc., producing however many trees required. Thus, if the user wishes to analyze a *collection* of functions, e.g., breakfns, he can perform (PRINTSTRUCTURE BREAKFNS).

If x is not a list, but is the name of a function, printstructure[x] is the same as printstructure[(x)]. Finally, if the value of x is a list of functions, printstructure will process that list as described above.

* NIL: is a TENEX output device that acts like a 'bottomless pit'. Note that file=NIL (not NIL:) means print tree to primary output file.

Note that in the case that x is a list, or evaluates to a list, subsequent functions are *not* separately analyzed if they have been encountered in the analysis of a function appearing earlier on the list. Thus the ordering of x can be important. For example, if both FOO and FIE call FUM, printstructure[(FOO FIE FUM)], will produce a tree for FOO containing embedded in it the tree for FUM. FUM will not be expanded in the tree for FIE, nor will it have a tree of its own. (Of course, if FOO also calls FIE, then FIE will not have a tree either.) The convention of listing FUM can be used to *force* printstructure to give FUM a tree of its own. Thus printstructure[(FOO FIE (FUM))] will produce *three* trees, and neither of the calls to FUM from FOO or FIE will be expanded in their respective trees. Of course, in this example, the same effect could have been achieved by reordering, i.e., printstructure[(FUM FOO FIE)]. However, if FOO, FIE, and FUM, all called each other, and yet the user wanted to see three separate trees, no ordering would suffice. Instead, the user would have to do printstructure[((FOO) (FIE) (FUM))].

The result of the analysis of printstructure is in two parts: donelst, a list summarizing the argument/variable information for each function appearing in the tree(s), and treelst, a list of the trees. last-printstructure is set to cons[donelst;treelst].

donelst is a list consisting, in alternation, of the functions appearing in any tree, and a variable list for that function. car of the variable list is a list of variables bound in the function, and cdr is a list of those variables used freely in the function. Thus the form of donelst for the earlier example would be:

```
(PRINTSTRUCTURE ((X FILE DONELST N TREELST TREEFNS L TEM X Y Z
FN TREE) PRDEPTH LAST-PRINTSTRUCTURE) PRGETD ((X FLG))
PROGSTRUC (( FN DEF N Y Z CALLSFLG VARSFLG VARS1 VARS2 D X)
N DONELST) ... ALLCALLS1 ((FN TR A B)))
```

Possible parentheses errors are indicated on donelst by a non-atomic form appearing where a function would normally occur, i.e., in an odd position. The non-atomic form is followed by the name of the function in which the P.P.E. occurred.

Printstructure Functions

printstructure[x;file]	analyzes x, saves result on last-printstructure, outputs trees and variable information to file, and returns x as its value.
treeprint[x;n]	prints a tree in the horizontal fashion shown in the examples above. i.e., printstructure performs (MAPC TREELST (FUNCTION TREEPRINT))
varprint[donelst;treelst]	prints the "lower half" of the printstructure output.
varprint1[fn;vars]	prints the variable information for a single function, e.g., (VARPRINT1 (CAR DONELST) (CADR DONELST)) produces first line of varprint output.
varprint2[fn;treelst]	prints the functions that call fn. e.g., (VARPRINT2 (CAR DONELST) TREELST) produces the second line of varprint output.
allcalls[fn;treelst]	uses treelst to produce a list of the functions that call fn.

firstfn[fn]

If *fn*=T, lower boundary is set to 0, i.e., all subrs and 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 *fn* is the name of a compiled function and the boundary is set at *fn*, i.e., all compiled functions defined earlier than *fn* are rejected.

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*, i.e., all compiled functions defined later than *fn* are rejected.

Thus to accept all compiled functions, perform firstfn[T] and lastfn[NIL], to accept no compiled functions, perform firstfn[].

calls[fn;varsflg]

is a fast 'one-level' *printstructure*, i.e., it indicates what functions *fn* calls, but does not go further and analyze any of them. *calls* does not print a tree, but 'reports its findings by returning as its value a list of three elements: a list of all functions called by *fn*, a list of variables bound in *fn*, and a list of variables used freely in *fn*, e.g.,

```
calls[progstruc]=
((PRGETD EXPRP PRGSTRC CCODEP
    CALLS1 ATTACH)
    (FN DEF N Y Z
  CALLSFLG VARSFLG VARS1 VARS2
    D X) (N DONELST))
```

fn can be a function name, a definition, or a form. Calls first does firstfn(T), lastfn() so that all subrs and compiled functions appear, except those on nofns or specifically eliminated as described in footnote on page 20.3. If varsflg is T, calls ignores functions and only looks at the variables (and therefore runs much faster).

notfn[fn]	Value of T indicates fn is *not* interesting. See discussion on pp. 20.3-20-4.
vars[fn]	cdr[calls[fn;T]]
freevars[fn]	cadr[vars[fn]]

CHAPTER XXI

MISCELLANEOUS

Contents

time[timex;timen;timetyp] is an nlambda function. It executes the computation timex, and prints out the number of conses and computation time. Garbage collection time is subtracted out.

```
TIME((LOAD (QUOTE PRETTY) (QUOTE PROP]
FILE CREATED  7-MAY-71 12:47:14

GC: 8
582, 10291 FREE WORDS
PRETTYFNS
PRETTYVARS
3727 CONSES
10.655 SECONDS
PRETTY
```

If timen is greater than 1 (timen=NIL equivalent to timen=1), executes timex timen number of times and prints out number of conses/timen, and computation time/timen. This is useful for more accurate measurement on small computations.

```
←TIME((COPY (QUOTE (A B C))) 10)
30/10 = 3 CONSES
.055/10 = .0055 SECONDS
(A B C)
```

If timetype is 0, time measures and prints total *real* time as well as computation time.

```
←TIME((LOAD (QUOTE PRETTY) (QUOTE PROP)) 1 0]
FILE CREATED  7-MAY-71 12:47:14

GC: 8
582, 10291 FREE WORDS
PRETTYFNS
PRETTYVARS
3727 CONSES
11.193 SECONDS
27.378 SECONDS, REAL TIME
PRETTY
```

If timetype = 3, time measures and prints garbage collection time as well as computation time.

```
←TIME((LOAD (QUOTE PRETTY) (QUOTE PROP)) 1 3]
FILE CREATED  7-MAY-71 12:47:14

GC: 8
582, 10291 FREE WORDS
PRETTYFNS
PRETTYVARS
3727 CONSES
10.597 SECONDS
1.487 SECONDS, GARBAGE COLLECTION TIME
PRETTY
```

Another option is timetype=T, in which case time measures and prints the number of pagefaults.

The value of time is the value of the last evaluation of timex.

date[ ]	Obtains date and time from TENEX and returns it as single string in format "dd-mmm-yy hh:mm:ss". where dd is day, mmm is month, yy year, hh hours, mm minutes, ss seconds, e.g., "14-MAY-71 14:26:08".
clock[n]	for n=0 current value of the time of day clock, i.e., number of milli-seconds since last system start up.
	for n=1 value of the time of day clock when the user started up this LISP, i.e., dif-ference between clock[Ø] and clock[1] is number of milli-seconds (real time) since this LISP was started.
	for n=2 number of milliseconds of *compute* time since user started up this LISP (garbage collection time is subtracted off)
	for n=3 number of milliseconds of compute time spent in gar-bage collections (all types).

dismiss[n]	dismisses program for *n* milliseconds, during which time program is in a state similar to an I/O wait, i.e., it uses no CPU time. Can be aborted by control-D, control-E, or control-B.
conscount[]	number of conses since this LISP started up. If given an argument (a number), resets *conscount* to this number.
gctrp[]	number of conses to next GC: 8, i.e., number of list words not in use. Note that an intervening GC of another type could collect as well as allocate additional list words. See section 3.
	gctrp[n] can be used to cause an interrupt when value of gctrp[]=n, see p. 10.16.
pagefaults[]	number of page faults since LISP started up.
logout[]	returns to TENEX. A subsequent CONTINUE command will enter the LISP program, return NIL as the value of the call to *logout*, and continue the computation exactly as if nothing had happened, i.e., *logout* is a programmable control-C. As with control-C, a REENTER command following a *logout* will reenter LISP at the top level.
	logout[] will not affect the state of any open files.

BREAKDOWN

Time gives analyses by computations. Breakdown is available to analyze the breakdown of computation time (or any other measure-able quantity) function by function. 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 COMMENT1)
(SUPERPRINT SUBPRINT COMMENT1)
←PRETTYDEF((SUBPRINT) FOO)
(SUBPRINT)
←RESULTS()
FUNCTIONS     TIME       # CALLS
SUPERPRINT    25.294        458
SUBPRINT      32.96         169
COMMENT1      7.833         12
TOTAL         66.087        639
NIL
```

The procedure used for measuring is such that if one function calls another and both are 'broken down', then the time (or whatever quantity is being measured) spent in the inner function is *not* charged to the outer function as well.

To remove functions from those being monitored, simply unbreak the functions, thereby restoring them to their original state. To add functions, call breakdown on the new functions. This will not reset the counters for any functions not on the new list. However breakdown[] can be used for zeroing the counters of all functions being monitored.

* This is with an interpreted prettyprint.

To use breakdown for some other statistic, before calling breakdown, set the variable brkdwntype to the quantity of interest, e.g., TIME, CONSES, etc. Whenever breakdown is called with brkdwntype not NIL, 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 may be called to compile the measuring function.* When breakdown is through initializing, it sets brkdwntype back to NIL. Subsequent calls to breakdown will measure the new statistic until brkdwntype is again set and a new breakdown performed. Sample output is shown below:

```
←SET(BRKDWNTYPE CONSES)
CONSES
←BREAKDOWN(MATCH CONSTRUCT)
(MATCH CONSTRUCT)
←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
```

The value of brkdwntype is used to search the list brkdwntypes for the information necessary to analyze this statistic. The entry on brkdwntypes corresponding to brkdwntype should be of the form (type form function), where form computes the statistic, and function (optional) converts the value of form to some more interesting quantity e.g. (TIME (CLOCK 2) (LAMBDA (X) (FQUOTIENT X 1000))) measures computation time and reports the result in seconds instead of milliseconds. If brkdwntype is not defined on brkdwntypes, an error is generated. brkdwntypes currently contains entries for TIME, CONSES, and PAGEFAULTS.

* The measuring functions for TIME and CONSES have already been compiled.

More Accurate Measurement

Occasionally, a function being analysed is sufficiently fast that the overhead involved in measuring it obscures the actual time spent in the function. If the user were using time, he would specify a value for timen greater than 1 to give greater accuracy. A similar option is available for breakdown. The user can specify that a function(s) be executed a multiple number of times for each measurement, and the average value reported, by including a number in the list of functions given to breakdown, e.g., BREAKDOWN(EDITCOM EDIT4F 10 EDIT4E EQP) means normal breakdown for editcom and edit4f but execute (the body of) edit4e and eqp 10 times each time they are called. Of course, the functions so measured must not cause any side effects, for obvious reasons. The printout from results will look the same as though each function were run only once, except that the measurement will be more accurate.

Subsys

This section describes a function, subsys, which permits the user to run a TENEX subsystem, such as SNDMSG, SRCCOM, TECO, or even another LISP, from inside of a LISP without destroying it. In particular, SUBSYS(EXEC) will start up a lower exec, which will print BBN TENEX ..., followed by @. The user can then do anything at this exec level that he can at the top level, without affecting his superior LISP. For example, he can start another LISP, perform a sysin, run for a while, type a control-C returning him to the lower exec, RESET, do a SNDMSG, etc. The user exits from the lower exec via the command QUIT, which will return control to subsys in the higher LISP. Thus with subsys, the user need not perform a sysout to save the state of his LISP in order to use a TENEX capability which would otherwise clobber the core image. Similarly, subsys provides a way of checking out a sysout file in a fresh LISP without having to commandeer another teletype or detach a job.

While subsys can be used to run any TENEX subsystem directly, without going through an intervening exec, this procedure is not recommended. The problem is that control-C always returns control to the next highest *exec*. Thus if the user is running a LISP in which he performs SUBSYS(LISP), and then types control-C to the lower LISP, control will be returned to the exec above the first LISP. The natural REENTER command would then clear the lower LISP,* but any files opened by it would remain open (until the next @RESET). If the user elects to call a subsystem directly, he must therefore know how it is normally exited and always exit from it that way.**

---

*A CONTINUE command however will return to the subordinate program, i.e. control-C followed by CONTINUE is safe at any level.

**LISP is exited via the function logout, TECO via the command ;H, SNDMSG via control-Z, and EXEC via QUIT.

Starting a lower exec does not have this disadvantage, since it can *only* be exited via QUIT, i.e., the lower exec is effectively 'errorset protected' against control-C.

Once control is returned to subsys and the higher LISP, a lower subsystem cannot be continued, i.e. its internal state is irretrievably lost. For example, if the user performs SUBSYS(LISP), runs for a while, and then does LOGOUT(), the lower LISP is lost. However, if instead he performs SUBSYS(EXEC), calls LISP from the lower exec, runs for awhile, and then does LOGOUT(), he *can* REENTER the lower LISP.

subsys[system;if;of]

If system=EXEC, starts up a lower exec, otherwise runs <SUBSYS>system. Control-C always returns control to next higher exec. Note that more than one LISP can be stacked, but there is no backtrace to help you figure out where you are.

if and of provide a way of specifying files for input and output. However, these must be used with extreme care as they can easily hang up your job in a compute bound state requiring divine intervention (i.e. R.S. Tomlinson or D.L. Murphy).

Edita

Edita is an editor for arrays. However, its most frequent application is in editing compiled functions (which are also arrays in BBN-LISP), and a great deal of effort in implementing edita, and most of its special features, are in this area. For example, edita knows the format and conventions of LISP compiled code, and so, in addition to decoding instructions a la DDT[†], edita can fill in the appropriate COREVALS, symbolic names for index registers, references to literals, linked function calls, etc. The following output shows a sequence of instructions in a compiled function first as they would be printed by DDT, and second by edita.

```
241456/    HRRZ 1,LISP&KNIL          3/    HRRZ 1,KNIL
241457/    HRL 1,242534              4/    HRL 1,'BRKZ
241460/    MOVEM 1,1(16)             5/    MOVEM 1,1(PP)
241461/    HRRZ 1,-10(16)            6/    HRRZ 1,<BRKCOM>(PP)
241462/    CAME 1,242535             7/    CAME 1,'↑
241463/    JRST 241474               8/    JRST 17                  ††
241464/    HRRZ 1,242536             9/    HRRZ 1,'BREAK1
241465/    HRRZM 1,2(16)            10/     HRRZM 1,2(PP)
241466/    HRRZ 1,242537            11/     HRRZ 1,'(ERROR!)
241467/    HRRZM 1,3(16)            12/     HRRZM 1,3(PP)
241470/    MOVE 1,LISP&MHC+ 5       13/     MOVE 1,740
241471/    MOVE 2,242541            14/     MOVE 2,';RETEVAL;
241472/    XCT @242520              15/     XCT @'8058306 56
241473/    JRST 242522              16/     JRST 551
241474/    CAME 1,242542            17/     CAME 1,'GO
241475/    JRST 241515              18/     JRST 34
241476/    HRRZ 1,@-6(16)           19/     HRRZ 1,@<BRKEXP>(PP)
```

Therefore, rather than presenting edita as an array editor with some extensions for editing compiled code, we prefer to consider it as a facility for editing compiled code, and point out that it can also be used for editing arbitrary arrays.

[†] DDT is one of the oldest debugging systems still around. For users unfamiliar with it, let us simply say that edita was patterned after it because so many people are familiar with it.

[††] Note that edita prints the addresses of cells contained *in* the function relative to the origin of the function.

Overview

To the user, edita looks very much like DDT with LISP extensions. It is a function of one argument, the name of the function to be edited.† Individual registers or cells in the function may be examined by typing their address followed by a slash,†† e.g.

```
                6/    HRRZ 1,<BRKCOM>(PP)
```

The 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 changed.* To change the contents of a register, the user first opens it, types the new contents, and then closes the register with a carriage return,†††† e.g.

```
              7/    CAME 1,'↑      CAMN 1,'↑↵
```

If the user closes a register without specifying the new contents, the contents are left unchanged. Similarly, if an error occurs or the user types control-E, the open register, if any, is closed without being changed.

---

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

††Underlined characters were typed by the user. edita uses its own read program, so that it is unnecessary to type a space before the slash, or to type a carriage return after the slash.

†††edita also converts absolute addresses of cells within the function to relative address on input. Thus, if the definition of foo begins at 85660, typing 6/ is *exactly* the same as typing 85666/.

††††Since carriage return has a special meaning, edita indicates the balancing of parentheses by typing a space.

Input Protocol

Edita processes all inputs not recognized as commands in the same way. If the input is the name of an instruction (i.e. an atom with a numeric OPD property), the corresponding number is added to the input value being assembled,[†] and a flag is set which specifies that the input context is that of an instruction.

The general form of a machine instruction is (opcode ac , @ address (index)) as described on p. 18.42. Therefore, in instruction context, edita evaluates all atoms (if the atom has a COREVAL property, the value of the COREVAL is used), and then if the atom corresponds to an ac ,[††] shifts it left 23 bits and adds it to the input value , otherwise adds it directly to the input value, but performs the arithmetic in the low 18 bits.[†††] Lists are interpreted as specifying index registers, and the value of car of the list (again COREVALs are permitted) is shifted left 18 bits. Examples:

```
HRRZ 1,KNIL
MOVEM 1,1(PP)
MOVE 1,733
HRRZ 1,@<BRKEXP>(PP)
XCT @ ';BREAKCOM1; 1
JRST 53 ORG
```

[†]The input value is initially Ø.

[††]i.e. if a ',' has not been seen, *and* the value of the atom is less than 16, *and* the low 18 bits of the input value are all zero.

[†††]If the absolute value of the atom is greater than 1000000Q, full word arithmetic is used. For example, the indirect bit is handled by simply binding @ to 20000000Q.

[††††]edita cannot in general know whether an address field in an instruction that is typed in is relative or absolute. Therefore, the user must add ORG, the origin of the function, to the address field himself. Note that edita would *print* this instruction, JRST 53 ORG, as JRST 53.

The user can also specify the address of a literal via the ' command (see p.21.17). For example, if the literal " UNBROKEN" is in cell 85672, HRRZ 1,'" UNBROKEN" is equivalent to HRRZ 1, 85672. Similarly, the user can specify a variable reference by enclosing the variable name in <> (see p.21.17). edita will compute the variables address relative to PP from its position in the literal table.[†] (See 6/ and 19/ in output on p. 21.10.)

When the input context is *not* that of an instruction, i.e. no OPD has been seen, all inputs are evaluated (the value of an atom with a COREVAL property is the COREVAL.) Then numeric values are simply added to the previous input value; non-numeric values *become* the input value.[††]

The only exception to the entire procedure occurs when a register is open that is in the pointer region of the function, i.e. literal table. In this case, atomic inputs are *not* evaluated. For example, the user can change the literal FOO to FIE by simply opening that register and then typing FIE followed by carriage return, e.g.

'FOO/   FOO   FIE↲

Note that this is equivalent to 'FOO/ FOO (QUOTE FIE)↲

[†]Note that the user must still type (PP), and also @ if the variable is a free variable .

[††]Presumably there is only one input in this case.

Commands and Variables

↵ (carriage return)	If a register is open and an input was typed, store the input in the register and close it.† If a register is open and nothing was typed, close the register without changing it. If a register is not open and input was typed, type its value.
ORG	Has the value of the address of the first instruction in the function. i.e. loc of getd of the function.
/	Opens the register specified by the low 18 bits of the quantity to the left of the /, and types its contents. If nothing has been typed, it uses the last thing typed by edita, e.g. 35/ JRST 53 / CAME 1,'RETURN / RETURN If a register was open, / closes it without changing its contents. After a / command, edita returns to that state of no input having been typed.

†If the register is in the unboxed region of the function, the unboxed value is stored in the register.

tab (control - I)	Same as carriage return, followed by the address of the quantity to the left of the tab, e.g.

```
35/     JRST 53    tab
53/     CAME 1,'RETURN
```

Note that if a register was open and input was typed, tab will change the open register before closing it, e.g.

```
35/     JRST 53    JRST 54  tab
54/     JRST 70    ↵
35/     JRST 54
```

. (period)	has the value of the address of the current (last) register examined.
line-feed	same as carriage return followed by (ADD1 .)/ i.e. closes any open register and opens the *next* register.
↑	same as carriage return followed by (SUB1 .)/
$Q (alt-modeQ)	has as its value the last quantity typed by edita e.g.

```
35/     JRST 53    $Q ^ ↵
./      JRST 54
```

LITS	has as value the (relative) address of the first literal.
BOXED	same as LITS
$ (dollar)	has as value the relative address of the last literal in the function.
=	Sets radix to -8 and types the quantity to the left of the = sign, i.e. if anything has been typed, it types the input value, otherwise, it types $Q, e.g.

```
 35/    JRST 54   =254ØØØ241541Q   JRST 54=254ØØØØØØØ66Q
```

Following =, radix is restored and edita returns to the no input state.

OK	leave edita
?	return to 'no input' state. ? is a 'weak' control-E, i.e. it negates any input typed, but does not close any registers.
address1, address2/	prints[†] the contents of registers address1 through address2. . is set to address2 after the completion.

[†]output goes to file, initially set to T. The user can also set file (while in edita) to the name of a disc file to redirect the output. (The user is responsible for opening and closing file.) Note that file only affects output for the address1, address2/ command.

'x	corresponds to the ' in LAP, p. 18.44. The next expression is read, and if it is a small number, the appropriate offset is added to it. Otherwise, the literal table is searched for x, and the value of 'x is the (absolute) address of that cell.[†] An error is generated if the literal is not found, i.e. ' cannot be used to *create* literals.
<atom>	The literal table is searched for atom, and the value of <atom> is the appropriate address relative to PP, e.g. if X is the last variable, <X> is 0, if X is the next to last variable, <X> is -1, etc. (If x is not a local or free variable, an error is generated).

[†] x of the form ;atom; specifies a linked function call.

$W (alt-modeW)   search command.

Searching consists of comparing the object of the search with the contents of each register, and printing those that match, e.g.

```
HRRZ @ $W
19/     HRRZ 1,@<BRKEXP>(PP)
40/     HRRZ 1,@<BRKVALUE>(PP)
72/     HRRZ 1,@<BRKCOMS>(PP)
122/     HRRZ 1,@<BRKFN>(PP)
216/     HRRZ 1,@<BRKFN>(PP)
345/     HRRZ 1,@<BRKEXP>(PP)
480/     HRRZ 1,@<BRKCOMS>(PP)
```

The $W command can be used to search either the unboxed portion of a function,i.e. instructions, or the pointer region, i.e. literals, depending on whether or not the object of the search is a number. If any input was typed before the $W, it will be the target, object of the search, otherwise the next expression is read and used as the object.[†] The user can specify a starting point for the search by typing an address followed by a ',' before calling $W, e.g. 1, JRST $W. If no starting point is specified, the search will begin at Ø if the object is a number, otherwise at LITS, the address of the first literal.[††] After the search is completed, '.' is set to the address of the last register that matched.

[†]Note that inputs typed before the $W will have been processed according to the input protocol,i.e. evaluated; inputs typed after the $W will not. Therefore, the latter form is usually used to specify searching the literals, e.g. $W FOO is equivalent to (QUOTE FOO) $W.

[††]Thus the only way the user can search the pointer region for a number is to specify the starting point via ','.

If the search is operating in the instruction region of the function, only those fields (i.e. instruction, ac, indirect, index, and address) of the object that contain one bits are compared.† For example, HRRZ @ $W will find all instances of HRRZ indirect, regardless of ac, index, and address fields. Similarly, 'PRINT $W will find all instructions that reference the literal PRINT.††

If the search is operating in the pointer region, a 'match' is as defined in the editor, p. 9.24. For example, $W (&) will find all registers that contain a list consisting of a single expression.

$C (alt-modeC)	like $W except only prints the first match, then prints the number of matches when the search finishes.

†Alternatively, the user can specify his own mask by setting the variable mask (while in edita), to the appropriate bit pattern.

††The user may need to establish instruction context for input without giving a specific instruction. For example, suppose the user wants to find all instructions with ac=1 and index=PP. In this case, the user can give & as a pseudo-instruction, e.g. type & 1, (PP).

:atom

defines atom to an address

(1) the value of $Q if a register is open,

(2) the input if any input was typed, otherwise

(3) the value of '.'.[†]

For example:

```
35/   JRST 54    :FOO
:FIE
FIE/    JRST FOO    .=35
```

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) and is used for *decoding* addresses. Usersyms is initially NIL, while symlst is set to a list of all the corevals. Since the : command adds the appropriate information to both 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.

[†]Only the low 18 bits are used and converted to relative addresses whenever possible.

Editing Arrays

Edita is called to edit a function by giving it the name of the function. Edita can also be called to edit an array by giving it the array as its first argument,† in which case the following differences are to be noted:

1. decoding - The contents of registers in the unboxed region are boxed and printed as numbers, i.e. they are never interpreted as instructions.

2. addressing convention - Whereas 0 corresponds to the first instruction of a function, the first element of an array by convention is element number 1.

3. input protocols - If a register is open, lists are evaluated, atoms are not evaluated (except for $Q which is always evaluated). If no register is open, all inputs are evaluated, and if the value is a number, it is added to the 'input value'.

4. left half - If the left half of an element in the pointer region of an array is not all Ø's or NIL, it is printed followed by a ;, e.g.

```
10/    (A B) ; T
```

Similarly, if a register is closed, either its left half, right half, or both halves can be changed, depending on the presence or absence, and position of the ; e.g.

```
10/   (A B); T   (A C);↵         changes left
 ./   (A C); T   NIL ↵           changes right
 ./   (A C); NIL  A ; C↵         changes both
 ./     A ; C
```

The $W command will look at both halves of elements in the pointer region.

---

†the array itself, *not* a variable whose value is an array.

SECTION XXII

THE PROGRAMMER'S ASSISTANT AND LISPX

Contents

Introduction

This chapter describes one of the newer additions to BBN-LISP: the programmer's assistant. The central idea of the programmer's assistant is that the user, rather than talking to a passive system which merely responds to each input and waits for the next, is instead addressing an active intermediary, namely his assistant. Normally, the assistant is invisible to the user, and simply carries out the user's requests. However, since the assistant remembers what the user has told him, the user can instruct him to repeat a particular operation or sequence of operations, with possible modifications, or to undo the effect of certain specified operations. Like DWIM, the programmer's assistant is not implemented as a single function or group of functions, but is instead dispersed throughout much of BBN-LISP.[†] Like DWIM, the programmer's assistant embodies a philosophy and approach to system design

[†]Some of the features of the programmer's assistant have been described elsewhere, e.g. the UNDO command in the editor, the file package, etc.

whose ultimate goal is to construct a programming environment which would "cooperate" with the user in the development of his programs, and free him to concentrate more fully on the conceptual difficulties and creative aspects of the problem he is trying to solve.

Example

The following dialogue, taken from an actual session at the console, gives the flavor of the programmer's assistant facility in BBN-LISP. The user is about to edit a function <u>loadf</u>, which contains several constructs of the form (PUTD FN2 (GETD FN1)). The user plans to replace each of these by equivalent MOVD expressions.

```
←EDITF(LOADFF]                                              [1]
=LOADF
EDIT
*PP
  [LAMBDA (X Y)
    [COND
      ((NULL (GETD (QUOTE READSAVE)))
        (PUTD (QUOTE READSAVE)
              (GETD (QUOTE READ]
    (PUTD (QUOTE READ)
          (GETD (QUOTE REED)))
  (NLSETQ (SETQ X (LOAD X Y)))
  (@UTD (QUOTE READ)
          (GETD (QUOTE READSAVE)))
    X]
*F PUTD (1 MOVD)                                            [2]
*3 (XTRR 2)                                                 [3]
=XTR
*ØP                                                         [4]
=Ø P
(MOVD (QUOTE READSAVE) (QUOTE READ))
*(SW 2 3)                                                   [5]
*
```

At [1], the user begins to edit <u>loadf</u>.† At [2] the user finds PUTD and replaces it by MOVD. He then shifts context to the third subexpression, [3], extracts its second subexpression, and ascends one level [4] to print the result. The user now switches the second and third subexpression [5], thereby completing the operation for this PUTD. Note that up to this point, the user has not directly addressed the assistant. The user now requests that the assistant print out the operations that the user has performed, [6], and the user then instructs the assistant to REDO FROM F, [7], meaning repeat the entire sequence of operations 15 through 20. The user then prints the current epxression, and observes that the second PUTD has now been successfully transformed.

```
*?? FROM F                                                  [6]

15.   *F PUTD
16.   *(1 MOVD)
17.   *3
18.   *(XTR 2)
19.   *Ø
2Ø.   *(SW 2 3)

*REDO FROM F                                                [7]
*P
(MOVD (QUOTE REED) (QUOTE READ))
*
```

†We prefer to consider the programmer's assistant as the moving force behind this type of spelling correction (even though the program that does the work is part of the DWIM package). Whereas correcting @RINT to PRINT, or XTRR to XTR does not require any information about what *this* user is doing, correcting LOADFF to LOADF clearly required noticing when this user defined <u>loadf</u>.

The user now asks the assistant to replay the last three steps to him, [8]. Note that the entire REDO FROM F operation is now grouped together as a single unit, [9], since it corresponded to a single user request. Therefore, the user can instruct the assistant to carry out the same operation again by simply saying REDO. This time a problem is encountered [10], so the user asks the assistant what it was trying to do [11].

```
*?? FROM -3                                          [8]

19.   *Ø
2Ø.   *(SW 2 3)
21.   REDO FROM F                                    [9]
      *F PUTD
      *(1 MOVD)
      *3
      *(XTR 2)
      *Ø
      *(SW 2 3)

*REDO

PUTD ?                                               [10]

?? -1                                                [11]

22.   REDO
      *F PUTD
      *(1 MOVD)
      *3
      *(XTR 2)
      *Ø
```

The user then realizes the problem is that the third PUTD is misspelled in the definition of LOADF. (see p. 22.2). He therefore instructs the assistant to USE @UTD FOR PUTD, [12], and the operation now concludes successfully.

```
*USE @UTD FOR PUTD                                      [12]
*P
(MOVD (QUOTE READSAVE) (QUOTE READ))
*↑ PP
  [LAMBDA (X Y)
    [COND
      ((NULL (GETD (QUOTE READSAVE)))
        (MOVD (QUOTE READ)
              (QUOTE READSAVE]
    (MOVD (QUOTE REED)
          (QUOTE READ))
    (NLSETQ (SETQ X (LOAD X Y)))
    (MOVD (QUOTE READSAVE)
          (QUOTE READ))
    X]
*OK
LOADF
←
```

An important point to note here is that while the user *could* have defined a macro to execute this operation, the operation is sufficiently complicated that he would want to try out the individual steps before attempting to combine them. At this point, he would already have executed the operation once. Then he would have to type in the steps again to define them as a macro, at which point the operation would only be repeated once more before failing. Then the user would have to repair the macro, or else change @UTD to PUTD by hand so that his macro would work correctly. It is far more natural to decide *after* trying a series of operations whether or not one wants them repeated or forgotten. In addition, frequently the user will think that the operation(s) in question will never need be repeated, and only discover afterwards that he is mistaken, as occurs when the operation was incorrect, but salvageable:

```
*P
(LAMBDA (STR FLGCQ VRB) **COMMENT** (PROG & & LP1 & LP2 & &))
*-1 -1 P
(RETURN (COND &))
*(-2 ((EQ BB (QUOTE OUT)) BB]                                [1]
*P
(RETURN (& BB) (COND &))                                     [2]
*UNDO
(-2 --) UNDONE
*2 P
(COND (EXPANS & & T))
*REDO EQ
*P
(COND (& BB) (EXPANS & & T)
*
```

Here the operation was correct, [1], but the context in which it was executed, [2], was wrong.

This example also illustrates one of the most useful functions of the programmer's assistant: its UNDO capability. In most systems, if a user suspected that a disaster might result from a particular operation, e.g. an untested program running wild and chewing up a complex data structure, he would prepare for this contingency by saving the state of part or all of his environment before attempting the operation. If anything went wrong, he would then back up and start over. However, saving/dumping operations are usually expensive and time consuming, especially compared to a short computation, and are therefore not performed that frequently, and of course there is always the case when disaster strikes as a result of a 'debugged' or at least innocuous operation, as shown in the following example:

```
←(MAPC ELTS (FUNCTION (LAMBDA (X) (REMPROP X (QUOTE MORPH]   [1]
NIL
←UNDO                                                           [2]
MAPC UNDONE.
←USE ELEMENTS FOR ELTS                                          [3]
NIL
←
```

The user types an expression which removes the property MORPH from every member of the list ELTS [1], and then realizes that he meant to remove that property only from those members of the list ELEMENTS, a much shorter list. In other words, he has deleted a lot of information that he actually wants saved. He therefore simply reverses the effect of the MAPC by typing UNDO [2], and then does what he intended via the USE command [3].

Overview

The programmer's assistant facility is built around a memory structure called the 'history list.' The history list is a list of the information associated with each of the individual 'events' that have occurred in the system, where each event corresponds to one user input.† For example, (XTR 2) ([3] on p. 22.2) is a single event, while REDO FROM F ([7] on p. 22.3) is also a single event, although the latter includes executing the operation (XTR 2), as well as several others.

Associated with each event on the history list is its input and its value, plus other optional information such as side-effects, formatting information, etc. If the event corresponds to a history command, e.g. REDO FROM F, the input corresponds to what the user would have had to type to execute the same operation(s), although the user's actual input, i.e. the history command, is saved in order to clarify the printout of that event ([9] on p. 22.4). Note that if a history command event combines several events, it will have more than one value:

†For various reasons, there are *two* history lists: one for the editor, and one for lispx, which processes inputs to evalqt and break, see p. 22.45.

```
←(LOG (ANTILOG 4))
4.Ø
←USE 4.Ø 4Ø 4ØØ FOR 4
4.Ø
4Ø.Ø
ARG NOT IN RANGE
4ØØ

←USE -4Ø.Ø -4.ØØØØ7 -19.
-4Ø.Ø
-4.ØØØØ7
-19.Ø
←USE LOG ANTILOG FOR ANTILOG LOG IN -2 AND -1
4.Ø
4Ø.Ø
4ØØ.Ø
4.ØØØØ7
19.Ø
←??


4.     USE LOG ANTILOG FOR ANTILOG LOG IN -2 -1
       ←(ANTILOG (LOG 4.Ø))
       4.Ø
       ←(ANTILOG (LOG 4Ø))
       4Ø.Ø
       (ANTILOG (LOG 4ØØ))
       4ØØ.Ø
       ←(ANTILOG (LOG -4Ø.Ø))
       4Ø.Ø
       ←(ANTILOG (LOG -4.ØØØØ7))
       4.ØØØØ7
       ←(ANTILOG (LOG -19.Ø))
       19.Ø
3.     USE -4Ø.Ø -4.ØØØØ7 -19.Ø
       ←(LOG (ANTILOG -4Ø.Ø))
       -4Ø.Ø
       ←(LOG (ANTILOG -4.ØØØØ7))
       -4.ØØØØ7
       ←(LOG (ANTILOG -19.Ø))
       -19.Ø
2.     USE 4.Ø 4Ø 4ØØ FOR 4
       ←(LOG (ANTILOG 4.Ø))
       4.Ø
       ←(LOG (ANTILOG 4Ø))
       4Ø.Ø
       ←(LOG (ANTILOG 4ØØ))

1.     ←(LOG (ANTILOG 4))
       4.Ø
```

As new events occur, existing events are aged, and the oldest event is 'forgotten.' For efficiency, the storage used to represent the forgotten event is cannibalized and reused in the representation of the new event, so the history list is actually a ring buffer. The size of this ring buffer is a system parameter called the 'time-slice.'[†] Larger time-slices enable longer 'memory spans,' but tie up correspondingly greater amounts of storage. Since the user seldom needs really 'ancient history,' and a NAME and RETRIEVE facility is provided for saving and remembering selected events, a relatively small time slice such as 30 events is more than adequate, although some users prefer to set the time slice as large as 100 events.

Events on the history list can be referenced in a number of ways. The output on p. 22.11 shows a printout of a history list with time-slice 16. The numbers printed at the left of the page are the event numbers. More recent events have higher numbers; the most recent event is event number 52, the oldest and about-to-be-forgotten event is number 37.[††] At this point in time, the user can reference event number 51, RECOMPILE(EDIT), by its event number, 51; its relative position, -2 (because it occurred two events back from the current time), or by a 'description' of its input, e.g. (RECOMPILE (EDIT)), or (& (EDIT)), or even just EDIT. As new events occur, existing events retain their absolute event numbers, although their relative positions change.

[†] Initially 30 events. The time slice can be changed with the function changeslice, p. 22.54.

[††] When the event number of the current event is 100, the next event will be given event number 1. (If the time slice is greater than 100, the 'roll-over' occurs at the next highest hundred, so that at no time will two events ever have the same event number. For example, if the time slice is 150, event number 1 follows event number 200.)

Similarly, descriptor references may require more precision to refer to an older event. For example, the description RECOMPILE would have sufficed to refer to event 51 had event 52, also containing a RECOMPILE, not intervened. Event specification will be described in detail later.

```
←??

52.   ... HIST UNDO
      ←RECOMPILE(HIST)
      HIST.COM
      ←RECOMPILE(UNDO)
      UNDO.COM
51.   ←RECOMPILE(EDIT)
      EDIT.COM
5Ø.   ←LOGOUT]

49.   ←MAKEFILES]
      (EDIT UNDO HIST)
48.   ←EDITF(UNDOLISPX)
      UNDOLISPX
47.   REDO GETD
      ←GETD(FIE)
      (LAMBDA (X) (MAPC X (F/L (PRINT X)))))
46.   ←UNDO
      FIE
45.   ←GETD(FIE)
      (LAMBDA (X) (MAPC X (FUNCTION (LAMBDA (X) (PRINT X)))))
44.   ←FIE]
      NIL
43.   ←DEFINEQ((FIE (LAMBDA (X) (MAPC X (F/L (PRINT X))))))
      (FIE)
42.   REDO GETD
      ←GETD(FIE)
      (LAMBDA (Y) Y)
41.   ←UNDO
      MOVD
4Ø.   REDO GETD
      ←GETD(FIE)
      (LAMBDA (X) X)
39.   ←MOVD(FOO FIE)
      FIE
38.   ←DEFINEQ((FOO (LAMBDA (X) X)))
      (FOO)
37.   ←GETD(FIE)
      (LAMBDA (Y) Y)
```

The most common interaction with the programmer's assistant occurs at the top level evalqt, or in a break, where the user types in expressions for evaluation, and sees the values printed out. In this mode, the assistant acts much like a standard LISP evalqt, except that before attempting to evaluate an input, the assistant first stores it in a new entry on the history list. Thus if the operation is aborted or causes an error, the input is still saved and available for modification and/or reexecution. The assistant also notes new functions and variables to be added to its spelling lists to enable future corrections. Then the assistant executes the computation (i.e. evaluates the form or applies the function to its arguments), saves the value in the entry on the history list corresponding to the input, and prints the result, followed by a prompt character to indicate it is again ready for input.†

If the input typed by the user is recognized as a history command, the assistant takes special action. Commands such as UNDO, ??, NAME, and RETRIEVE are immediately performed. Commands that involved reexecution of previous inputs, e.g. REDO and USE, are achieved by computing the corresponding input expression(s) and

---

†The function that accepts a user input, saves the input on the history list, performs the indicated computation or history command, and prints the result, is lispx. lispx is called by evalqt and breakl, and in most cases, is synonymous with 'programmer's assistant.' However, for various reasons, the editor saves its own inputs on a history list, carries out the requests, i.e. edit commands, and even handles undoing independently of lispx. The editor only calls lispx to execute a history command, such as REDO, USE, etc. Therefore we use the term assistant (loosely) when the discussion applies to features shared by evalqt, break and the editor, and the term lispx when we are discussing the specific function.

then *unreading* them. The effect of this unreading operation is to cause the assistant's input routine, lispxread, to act exactly as though these expressions were typed in by the user. Except for the fact that these inputs are not saved on new and separate entries on the history list, but associated with the history command that generated them, they are processed exactly as though they had been typed.

The advantage of this implementation is that it makes the programmer's assistant a callable facility for other system packages as well as for users with their own private executives. For example, breakl accept user inputs, recognizes and executes certain break commands and macros, and interprets anything else as LISP expressions for evaluation. To interface breakl with the programmer's assistant required three small modifications to breakl: (1) input was to be obtained via lispxread instead of read; (2) instead of calling eval or apply directly, breakl was to give those inputs it could not interpret to lispx, and (3) any commands or macros handled by breakl, i.e. not given to lispx, were to be stored on the history list by breakl by calling the function historysave, a part of the assistant package.

Thus when the user typed in a break command, the command would be stored on the history list as a result of (1). If the user typed in an expression for evaluation, it would be evaluated as before, with the expression and its value both saved on the history list as a result of (2). Now if the user entered a break and typed three inputs: EVAL, (CAR !VALUE), and OK, at the next break, he could achieve the same effect by typing REDO FROM EVAL. This would cause the assistant to unread the three expressions EVAL, (CAR !VALUE), and OK. The next 'input' seen by breakl would then be EVAL, which breakl would interpret. Next would come (CAR !VALUE), which would be given to lispx to evaluate, and then would come OK, which breakl would again process. Thus, by virtue of unreading, history operations will work even for those inputs not interpretable by lispx, in this case, EVAL and OK.

The net effect of this implementation of the programmer's assistant is to provide a facility which is easily inserted at many levels, and embodies a consistent set of commands and conventions for talking about past events. This gives the user the subjective feeling that a single agent is watching everything he does and says, and is always available to help.

Event Specification

All history commands use the same conventions and syntax for indicating which event or events on the history list the command refers to, even though different commands may be concerned with different aspects of the corresponding event(s), e.g. side-effects, value, input, etc. Therefore, before discussing the various history commands in the next section, this section describes the types of event specifications currently implemented. All examples refer to the history list on page 22.11.

An event address identifies one event on the history list. It consists of a sequence of 'commands' for moving an imaginary cursor up or down the history list, much in the manner of the arguments to the @ command in break (see pp. 15.8-15.9). The event identified is the one 'under' the imaginary cursor when there are no more commands. (If any command fails, an error is generated and the history command is aborted.)

The commands are interpreted as follows:

n (n>ϕ)	move forward n events, i.e. in direction of increasing event numbers. If given as the first 'command,' n specifies the event with event number n.
n (n<ϕ)	move backward -n events.
←atom	search backward for an event whose *function* matches atom (i.e. for apply format only), e.g. whereas FIE would refer to event 47, ←FIE would refer to event 44. Similarly, ED\$[†] would specify event 51, whereas ←ED\$ event 48.

[†]i.e. EDalt-mode.

pat†	search backward for an event whose input contains an expression that matches pat as described on p. 9.24-9.25 of the editor.
←	next search is to go forward instead of backward, (if given as the first 'command', next search begins with last, i.e. oldest, event on history list), e.g. ← LAMBDA refers to event 38; MAKEFILE ← RECOMPILE refers to event 51.
=	next search is to look at *values*, instead of inputs, e.g. = UNDO refers to event 49; 45 = FIE refers to event 43; ← = LAMBDA refers to event 37.
\	specifies the event last located.

Note: each search skips the current event, i.e. each command always moves the cursor. For example, if FOO refers to event n, FOO FIE will refer to some event before event n, even if there is a FIE in event n.

†i.e. anything else except for ←, =, and \ , which are interpreted as described above.

An <u>event specification</u> specifies one or more events:

FROM #1 THRU #2 #1 THRU #2	the sequence of events from the event with address #1 through event with address #2,[†] e.g. FROM GETD THRU 49 specifies events 47, 48, and 49. #1 can be more recent than #2, e.g. FROM 49 THRU GETP specifies events 49, 48, and 47 (note reversal of order).
FROM #1 TO #2 #1 TO #2	Same as THRU but does not include event #2.
FROM #1	Same as FROM #1 THRU -1, e.g. FROM 49 specifies events 49, 50, 51, and 52.
THRU #2	Same as FROM -1 THRU #2, e.g. THRU 49 specifies events 52, 51, 50, and 49. Note reversal of order.
TO #2	Same as FROM -1 TO #2
#1 AND #2 AND ... AND #n	i.e. a sequence of event addresses separated by AND's, e.g. FROM 47 TO LOGOUT would be equivalent to 47 AND 48 AND MAKEFILES.
empty	i.e. nothing specified, same as -1, unless last event was an UNDO, in which case same as -2.[††]
@ atom	refers to the events named by atom, via the NAME command, p.22.26. if the user names a particular event or events FOO, @ FOO specifies those events.
@@ ¢	¢ is an event specification and interpreted as above, but with respect to the archived history list, as specified on p. 22.28.

[†] i.e. the symbol #1 corresponds to all commands between FROM and THRU in the event specification, and #2 to all commands from THRU to the end of the event specification. For example, in FROM FOO 2 THRU FIE -1, #1 is (FOO 2), and #2 is (FIE -1).

[††] For example, if the user types (NCONC FOO FIE), he can then type UNDO, followed by USE NCONC1.

History Commands

All history commands can be input as either lists, or as lines, e.g. (REDO FROM -3) or REDO FROM -3. The latter format is easier to type, but note that the appearance of a list will terminate a line input (see readline p. 14.11). Thus, the command (USE (FOO) FOR FOO) can only be entered in list format.

¢ is used to denote an event specification. Unless specified otherwise, ¢ omitted is the same as ¢ = -1, e.g. REDO and REDO -1 are the same.

REDO ¢	redoes the event or events specified by ¢, e.g. REDO FROM -3 redoes the last three events.
USE vars FOR args IN ¢	substitutes vars for args in ¢, and redoes the result, e.g. USE LOG ANTILOG FOR ANTILOG LOG IN -2 AND -1. Substitution is carried out as described below.
USE vars_1 FOR args_1 AND vars_2 FOR args_2 AND ... AND vars_n FOR args_n IN ¢	More general form of USE command. See description of substitution algorithm below.

Every USE command involves three pieces of information: the variables to be substituted, the arguments to be substituted for, and an event specification, which defines the expression (input) in which the substitution takes place.[†]

If args are omitted, i.e. the form of the command is USE vars IN ¢, or just USE vars (which is equivalent to USE vars IN -1), and the event referred to was itself a USE command, the arguments and expression substituted into are the same as for the indicated USE command. In effect, this USE command is thus a continuation of the previous USE command. For example, on page 22.9, when the user types (LOG (ANTILOG 4)), followed by USE 4.0 40 400 FOR 4, followed by USE -40.0 -4.00007 -19., the latter command is equivalent to USE -40.0 -4.00007 -19. FOR 4 IN -2.

[†]The USE command is parsed by a small finite state parser to distinguish the variables and arguments. For example, USE FOR FOR AND AND AND FOR FOR will be parsed correctly.

If args are omitted and the event referred to was *not* a USE command, substitution is for the operator in that command, i.e. if a lispx input, the name of the function, if an edit command, the name of the command. For example ARGLIST(FOO) followed by USE CALLS is equivalent to USE CALLS FOR ARGLIST.

If ¢ is omitted, but args are specified, the first member of args is used for ¢, e.g. USE PUTD FOR @UTD is equivalent to USE PUTD FOR @UTD IN @UTD.[†]

If the USE command has the same number of variables as arguments, the substitution procedure is straightforward,[††] i.e. USE X Y FOR U V means substitute X for U and Y for V, and is equivalent to USE X FOR U AND Y FOR V. However, the USE command also permits distributive substitutions, i.e. substituting several variables for the same argument. For example, USE A B C FOR X means first substitute A for X then substitute B for X (in a new copy of the expression), then substitute C for X. The effect is the same as three separate USE commands. Similarly, USE A B C FOR D AND X Y Z FOR W is equivalent to USE A FOR D AND X FOR W, followed by USE B FOR D AND Y FOR W, followed by USE C FOR D AND Z FOR W. USE A B C FOR D AND X FOR Y[†††] also corresponds to three

---

[†]A special check is made for the case where the first arg is a number. e.g. USE 4.0 40 400 FOR 4. Obviously the user means find the event containing a 4 and perform the indicated substitutions, whereas USE 4.0 40 400 FOR 4 IN 4 would mean perform the substitutions in event *number* 4.

[††]Except when one of the arguments and one of the variables are the same, e.g. USE X Y FOR Y X, or USE X FOR Y AND Y FOR X. This situation is noticed when parsing the command, and handled correctly.

[†††]or USE X FOR Y AND A B C FOR D.

substitutions, the first with A for D and X for Y, the second with B for D, and X for Y, and the third with C for D, and again X for Y. However, USE A B C FOR D AND X Y FOR Z is ambiguous and will cause an error. Essentially, the USE command operates by proceeding from left to right handling each 'AND' separately. Whenever the number of variables exceeds the number of expressions available, the expressions multiply.[†]

FIX ¢ puts the user in the editor looking at a copy of the input(s) for ¢. When the user exits via OK, the result is unread and reexecuted exactly as with REDO.

FIX is provided for those cases when the modifications to the input(s) are not of the type that can be specified by USE, i.e. not substitutions. For example:

```
←(DEFINEQ FOO (LAMBDA (X) (FIXSPELL SPELLINGS2 X 70]

INCORRECT DEFINING FORM
FOO

←FIX
EDIT
*P
(DEFINEQ FOO (LAMBDA & &))
*(LI 2)
OK
(FOO)
←
```

[†]Thus USE A B C D FOR E F means substitute A for E at the same time as substituting B for F, then in another copy of the indicated expression, substitute C for E and D for F. Note that this is also equivalent to USE A C FOR E AND B D FOR F.

Implementation of REDO, USE, and FIX

The input portion of an event is represented internally on the history list simply as a linear sequence of the expressions which were read. For example, an input in apply format is a list consisting of two expressions, an input in eval format is a list of just one expression.[†] Thus if the user wishes to convert an input in apply format to eval format, he simply moves the function name inside of the argument list:

```
←MAPC(FOOFNS (F/L (AND (EXPRP X) (PRINT X]
NIL
←EXPRP(FOO1)
T
←FIX MAPC
EDIT
*P
(MAPC (FOOFNS &))
*(BO 2)
*(LI 1)
*P
((MAPC FOOFNS &))
OK
FOO1
FIE2
FUM
NIL
←
```

By simply converting the input from two expressions to one expression, the desired effect, that of mapping down the list that was the *value* of foofns, was achieved.

[†]For inputs in eval format, i.e. single expressions, FIX calls the editor so that the current expression is that input, rather than the list consisting of that input - see the example on the preceding page. However, the entire list is actually being edited. Thus if the user typed ↑ P in that example, he would see ((DEFINEQ FOO &)).

REDO, USE and FIX all operate by obtaining the input portion of the corresponding event, processing the input (except for REDO), and then storing it on the history list as the input portion of a new event. The history command completes operating by simply unreading the input. When the input is subsequently 'reread', the event which already contains the input will be retrieved and used for recording the value of the operation, saving side-effects, etc., instead of creating a new event. Otherwise the input is treated exactly the same as if it had been typed in directly.

If ¢ specifies more than one event, the inputs for the corresponding events are simply concatenated into a linear sequence, with special markers representing carriage returns† inserted between each input to indicate where new lines start. The result of this concatenation is then treated as the input referred to by ¢. For example, when the user typed REDO FROM F ([7] on p. 22.3) the inputs for the corresponding six events were concatenated to produce (F PUTD #0 (1 MOVD) #0 3 #0 (XTR 2) #0 0 #0 (SW 2 3)). Similarly, if the user had typed USE @UTD FOR PUTD IN 15 THRU 20, (F PUTD #0 (1 MOVD) #0 3 #0 (XTR 2) #0 0 #0 (SW 2 3)) would have been constructed, and then @UTD substituted for PUTD throughout it.

The same convention is used for representing multiple inputs when a USE command involves sequential substitutions. For example, if the user types GETD(FOO) and then USE FIE FUM FOR FOO, the input sequence that will be constructed is (GETD (FIE) #0 GETD (FUM)), which is the result of substituting FIE for FOO in (GETD (FOO)) concatenated with the result of substituting FUM for FOO in (GETD (FOO)).

†The value of (VAG 0) is currently used to represent a carriage return on the grounds that it cannot be typed in by the user, and thus cannot cause ambiguities.

Once such a multiple input is constructed, it is treated exactly the same as a single input, i.e. the input sequence is recorded in a new event, and then unread, exactly as described above. When the inputs are 'reread,' the 'pseudo-carriage-returns' are treated by lispxread and readline exactly as real carriage returns, i.e. they serve to distinguish between apply and eval formats on inputs to lispx, and to delimit line commands to the editor. Note that once this multiple input has been entered as the input portion of a new event, that event can be treated exactly the same as one resulting from type in. In other words, no special checks have to be made when *referencing* an event, to see if it is simple or multiple. Thus, when the user types REDO following REDO FROM F, ([10] p. 22.4) REDO does not even notice that the input retrieved from the previous event is (F PUTD #0 ... (SW 2 3)) i.e. a multiple input, it simply records this input and unreads it. Similarly, when the user then types USE @UTD FOR PUTD, the USE command simply carries out the substitution, and the result is the same as though the user had typed USE @UTD FOR PUTD IN 15 THRU 20.

In sum, this implementation permits ¢ to refer to a single simple event, or to several events, or to a single event originally constructed from several events (which may themselves have been multiple input events, etc.) without having to treat each case separately.

History Commands Applied to History Commands

Since history commands themselves do *not* appear in the input portion of events (although they are stored elsewhere in the event), they do not interfere with or affect the searching operations of event specifications. In effect, history commands are invisible to event specifications. As a result, history commands themselves cannot be recovered for execution in the normal way. For example, if the user types USE A B C FOR D and follows this with USE E FOR D, he will not produce the effect of USE A B C FOR E (but instead will simply cause E to be substituted for D in the last event containing a D). To produce this effect, i.e. USE A B C FOR E, the user should type USE D FOR E IN *USE*. The appearance of the word REDO, USE or FIX in an event address specifies a search for the corresponding history command. (For example, the user can also type UNDO REDO.) It also specifies that the text of the history command itself be treated as though it were the input. However, *the user must remember that the context in which a history command is reexecuted is that of the current history, not the original context.* For example, if the user types USE FOO FOR FIE IN -1, and then later types REDO USE, the -1 will refer to the event before the REDO, not before the USE. Similarly, if the user types REDO REDO followed by REDO REDO, he would cause an infinite loop, except for the fact that a special check detects this situation.

More History Commands

RETRY ¢	similar to REDO except sets helpclock so that any errors that occur while executing ¢ will cause breaks.
... vars	similar to USE except substitutes for the (first) *operand*.

For example, EXPRP(FOO) followed by ... FIE FUM is equivalent to USE FIE FUM FOR FOO. See also event 52 on page 22.11.

?? ¢	prints history list. If ¢ is omitted, ?? prints the entire history list, beginning with most recent events. Otherwise ?? prints only those events specified in ¢ (and in the order specified), e.g. ?? -1, ?? 10 THRU 15, etc.

?? commands are not entered on the history list, and so do not affect relative event numbers. In other words, an event specification of -1 typed following a ?? command will refer to the event immediately preceding the ?? command.

?? will print the history command, if any, associated with each event as shown on p. 22.4, [9] and 22.9. Note that these history commands are not preceded by prompt characters, indicating they are not stored as input.†

?? prints multiple input events under one event number (see p. 22.9).

Since events are initially stored on the history list with their value field equal to bell, i.e. control-G, if an operation fails to complete for any reason, e.g. causes an error, is aborted, etc., its 'value' will be bell. This is the explanation for the blank line in event 2, p. 22.9 and event 50, p. 22.11.

†REDO, RETRY, USE, ..., and FIX commands, i.e. those commands that reexecute previous events, are not stored as inputs, because the input portion for these events are the expressions to be 'reread'. The history commands UNDO, NAME, RETRIEVE, BEFORE, and AFTER *are* recorded as inputs, and ?? prints them exactly as they were typed.

UNDO ¢	undoes the side effects of the specified events. For each event undone, UNDO prints a message: e.g. RPLACA UNDONE, REDO UNDONE etc. If nothing is undone because nothing was saved, UNDO types NOTHING SAVED. If nothing was undone because the event(s) were already undone, UNDO types ALREADY UNDONE. If ¢ is empty, UNDO searches back for the last event that contained side effects, was not undone, and itself was not an UNDO command.† ††
NAME atom ¢	saves the event(s) (including side effects) specified by ¢ on the property list of atom e.g. NAME FOO 10 THRU 15. NAME commands are undoable.
RETRIEVE atom	Retrieves and reenters on the history list the events named by atom. Causes an error if atom was not named by a NAME command.

For example, if the user performs NAME FOO 10 THRU 15, and at some time later types RETRIEVE FOO, 6 *new* events will be recorded on the history list (whether or not the corresponding events have been forgotten yet). Note that RETRIEVE does *not* reexecute the events, it simply retrieves them. The user can then REDO, UNDO, FIX, etc. any or all of these events. Note that the user can combine the effects of a RETRIEVE and a subsequent history command in a single

†Note that the user can undo UNDO commands themselves by specifying the corresponding event address, e.g. UNDO -3 or UNDO UNDO.

††UNDOing events in the reverse order from which they were executed is guaranteed to restore all pointers correctly, e.g. to undo all effects of last five events, perform UNDO THRU -5, *not* UNDO FROM -5. Undoing out of order may have unforeseen effects if the operations are *dependent*. For example, if the user performed (NCONC1 FOO FIE), followed by (NCONC1 FOO FUM), and then undoes the (NCONC1 FOO FIE), he will also have undone the (NCONC1 FOO FUM). If he then undoes the (NCONC1 FOO FUM), he will cause the FIE to reappear, by virtue of restoring FOO to its state before the execution of (NCONC1 FOO FUM). For more details, see p. 22.43.

operation by using an event specification of the form @ atom, as described on page 22.17, e.g. REDO @ FOO is equivalent to RETRIEVE FOO, followed by an appropriate REDO.[†] Note that UNDO @ FOO and ?? @ FOO are permitted.

BEFORE atom	undoes the effects of the events named by <u>atom</u>.
AFTER atom	undoes a BEFORE <u>atom</u>.

BEFORE/AFTER provide a convenient way of flipping back and forth between two states, namely that state *before* a specified event or events were executed, and that state *after* execution. For example, if the user has a complex data structure which he wants to be able to interrogate before and after certain modifications, he can execute the modifications, name the corresponding events with the NAME command, and then can turn these modifications off and on via BEFORE or AFTER commands.[††] Both BEFORE and AFTER are NOPs if the <u>atom</u> was already in the corresponding state; both generate errors if <u>atom</u> was not named by a NAME command.

Note: since UNDO, NAME, RETRIEVE, BEFORE, and AFTER are recorded as inputs they can be referenced by REDO, USE, etc. in the normal way. However, the user must again remember that the context in which the command is reexecuted is different than the original context. For example, if the user types NAME FOO DEFINEQ THRU COMPILE, then types ... FIE, the input that will be reread will be NAME FIE DEFINEQ THRU COMPILE as was intended, but both DEFINEQ and COMPILE, will refer to the most recent event containing those atoms, namely the event consisting of NAME FOO DEFINEQ THRU COMPILE!

[†]Actually, REDO @ FOO is better than RETRIEVE followed by REDO since in the latter case, the corresponding events would be entered on the history list *twice*, once for the RETRIEVE and once for the REDO.

[††]The alternative to BEFORE/AFTER for repeated switching back and forth involves UNDO, UNDO of the UNDO, UNDO of that etc. At each stage, the user would have to locate the correct event to undo, and furthermore would run the risk of that event being 'forgotten' if he did not switch states at least once per time-slice.

ARCHIVE ¢ — records the events specified by ¢ on a permanent history list. This history list can be referenced by preceding a standard event specification with @@, e.g. ?? @@ prints the archived history list, REDO @@ -1 will recover the corresponding event from the archived history list and redo it, etc.

The user can also provide for automatic archiving of selected events by appropriately defining <u>archivefn</u>, as described on p. 22.34

FORGET ¢ — permanently erases the record of the side effects for the events specified by ¢. If ¢ is omitted, forgets side effects for entire history list.

FORGET is provided for users with space problems. For example, if the user has just performed <u>sets</u>, <u>rplacas</u>, <u>rplacds</u>, <u>putd</u>, <u>remprops</u>, etc. to release storage, the old pointers would not be garbage collected until the corresponding events age sufficiently to drop off the end of the history list and be forgotten. FORGET can be used to force immediate forgetting (of the side-effects only). FORGET is not undoable (obviously).

Miscellaneous Features and Commands

TYPE-AHEAD	is a command that allows the user to type-ahead an indefinite number of inputs.

The assistant responds to TYPE-AHEAD with a ready character of >. The user can now type in an indefinite number of lines of input, under errorset protection. The input lines are saved and unread when the user exits the type-ahead loop with the command $OK (alt-modeOK) or $GO (no difference between the two commands). While in the type-ahead loop, ?? can be used to print the type-ahead, FIX to edit the type-ahead, and $Q to erase the last input (may be used repeatedly). For example:

```
←TYPE-AHEAD
>SYSOUT(TEM)
>MAKEFILE(EDIT)
>BRECOMPILE((EDIT WEDIT))
>F
>$Q
\\F
>$Q
\\BRECOMPILE
>LOAD(WEDIT PROP)
>BRECOMPILE((EDIT WEDIT))
>F
>MAKEFILE(BREAK)
>LISTFILES(EDIT BREAK)
>SYSOUT(CURRENT)
>LOGOUT]
>??
     >SYSOUT(TEM)
     >MAKEFILE(EDIT)
     >LOAD(WEDIT PROP)
     >BRECOMPILE((EDIT WEDIT))
     >F                                  †
     >MAKEFILE(BREAK)
     >LISTFILES(EDIT BREAK)
     >SYSOUT(CURRENT)
     >LOGOUT]
>FIX
EDIT
*(R BRECOMPILE BCOMPL)
*P
((LOGOUT) (SYSOUT &) (LISTFILES &) (MAKEFILE &) (F) (BCOMPL &) (LOAD &)
 (MAKEFILE &) (SYSOUT &))
*(DELETE LOAD)
*OK
>$GO
```

The TYPE-AHEAD command may be aborted by $STOP; control-E simply aborts the current line of input.

---

†Note that type-ahead can be addressed to the compiler, since it uses lispxread for input. Type-ahead can also be directed to the editor, but type-ahead to the editor and to lispx cannot be intermixed.

* * *

$ (alt-mode)	is a command for recovering the input buffers.

Whenever an error occurs in executing a lispx input or edit command, or a control-E or control-D is typed, the input buffers are saved and cleared. The alt-mode command is used to restore the input buffers, i.e. its effect is exactly the same as though the user had retyped what was 'lost.' For example:

```
*(-2 (SETQ X (COND ((NULL Z) (CONS          (user typed control-E)
*P
(COND (& &) (T &))
*2
*$
(-2 (SETQ X (COND ((NULL Z) (CONS
```

and user can now finish typing the (-2 ..) command.

Note: the type-ahead does not have to be 'seen' by LISP, i.e. echoed, since the system buffer is also saved.

Input buffers are not saved on the history list, but on a free variable. Thus, only the contents of the input buffer as of the last clearbuf can ever be recovered. However, input buffers cleared at evalqt are saved independently from those cleared by break or the editor. The procedure followed when the user types $ is to recover first from the local buffer, otherwise from the top level buffer.† Thus the user can lose input in the editor, go back to evalqt, lose input there, then go back into the editor, recover the editor's buffer, etc. Furthermore, a buffer cleared at the top can be recovered in a break, and vice versa.

---

†The local buffer is stored on lispxbufs; the top level buffer on toplispxbufs. The forms of both buffers are (CONS (LINBUF) (SYSBUF)) (see p. 14.17). Recovery of a buffer is destructive, i.e. $ sets the corresponding variable to NIL. If the user types $ when both lispxbufs and toplispxbufs are NIL, the message NOTHING SAVED is typed, and an error generated.

⁑ ⁑ ⁑

The following four commands, DO, !F, !E, and !N, are only recognized in the editor:

DO com	allows the user to supply the command name when it was omitted. (USE is used when a command name is *incorrect*).

For example, suppose the user wants to perform (-2 (SETQ X (LIST Y Z))) but instead types just (SETQ X (LIST Y Z)). The editor will type SETQ ?, whereupon the user can type DO -2. The effect is the same as though the user had typed FIX, followed by (LI 1), (-1 -2), and OK, i.e. the command (-2 (SETQ X (LIST Y Z))) is executed. DO also works if the last command is a line command.

!F	same as DO F.

In the case of !F, the previous command is always treated as though it were a line command, e.g. if the user types (SETQ X &) and then !F, the effect is the same as though he had typed F (SETQ X &), not (F (SETQ X &)).

!E	same as DO E. Note !E works correctly for 'commands' typed in eval or apply format.
!N	same as DO N.

⁂ ⁂ ⁂

control-U	when typed in at any point during an input being read by lispxread, permits the user to edit the input before it is returned to the calling function.

This feature is useful for correcting mistakes noticed in typing *before* the input is executed, instead of waiting till after execution and then performing an UNDO and a FIX. For example, if the user types DEFINEQ(FOO (LAMBDA (X) (FIXSPELL X and at that point notices the missing left parenthesis, instead of completing the input and allowing the error to occur, and then fixing the input, he can simply type control-U,[†] finish typing normally, whereupon the editor is called on (FOO (LAMBDA (X) (FIXSPELL X -- , which the user can then repair, e.g. by typing (LI 1). If the user exits from the editor via OK, the (corrected) expression will be returned to whoever called lispxread exactly as though it had been typed.[††] If the user exits via STOP, the expression is returned so that it can be stored on the history list. However it will *not* be executed. In other words, the effect is the same as though the user had typed control-E at exactly the right instant.

[†] Control-U can be typed at any point, even in the middle of an atom; it simply sets an internal flag checked by lispxread.

[††] Control-U also works for calls to readline , i.e. for line commands.

⁘ ⁘ ⁘

valueof	is an nlambda function for obtaining the value of a particular event, e.g. (VALUEOF -1),[†] (VALUEOF ←FOO -2) The value of an event consisting of several operations is a list of the values for each of the individual operations. Note: the value field of a history entry is initialized to bell (control-G). Thus a value of bell indicates that the corresponding operation did not complete, i.e. was aborted or caused an error (or else returned bell).
prompt#flg	is a flag which when set to T causes the current event number to be printed before each ←,: and * prompt characters. See description of promptchar, p. 22.51. prompt#flg is initially NIL.

⁘ ⁘ ⁘

archivefn	allows the user to specify events to be automatically archived.

When archivefn is *set* to T, and an event is about to drop off the end of the history list and be forgotten, archivefn is called giving it as its first argument the input portion of the event, and as its second argument, the entire event.[††] If archivefn

[†]Although the input for valueof is entered on the history list before valueof is called, valueof[-1] still refers to the value of the expression immediately before the valueof input, because valueof effectively backs the history list up one entry when it retrieves the specified event. Similarly, (VALUEOF FOO) will find the first event before this one that contains a FOO.

[††]In case archivefn needs to examine the value of the event, its side effects, etc. See p. 22.45 for discussion of the format of history lists.

returns T, the event is archived. For example, some users like to keep a record of all calls to load. Defining archivefn as (LAMBDA (X Y) (EQ (CAR X) (QUOTE LOAD))) will accomplish this. Note that archivefn must be *both* set and defined. archivefn is initially NIL and undefined.

⁑ ⁑ ⁑

lispxmacros provides a macro facility for lispx.

lispxmacros allows the user to define his own lispx commands. It is a list of elements of the form (command def). Whenever command appears as the first expression on a line in a lispx input, the variable lispxline is bound to the rest of the line, the event is recorded on the history list, and def is evaluated. Similarly, whenever command appears as car of a form in a lispx input, the variable lispxline is bound to cdr of the form, the event recorded, and def is evaluated. (See p. 22.58 for an example of a lispxmacro).

⁑ ⁑ ⁑

userinputp provides a way for a user function to process selected inputs.

When userinputp is set to T, it is applied[†] to all inputs not recognized as one of the commands described above. If userinputp returns T, the input will be recorded on the history list, and when the time comes for 'evaluation,' it will be given to lispxuserfn instead of eval or apply.[††] Thus by appropriately defining userinputp and lispxuserfn (and setting userinputp), the user can with a minimum of effort incorporate the features of the programmer's assistant into his own executive program (actually it is the other way around).

[†]Like archivefn, userinputp must be both set and defined.

[††]The purpose of two separate functions, userinputp and lispxuserfn, one for testing and one for execution, is to free the user from having to save the input on the history list himself.

The following output illustrates such a coupling.†

```
**SETQ(ALTFORM (MAPCONC NASDIC (F/L (GETP X 'ALTFORMS]                  [1]
=NASDICT
(AL26 BE7 CO56 CO57 CO6Ø C13 H3 MN54 NA22 SC46 S34 TI44)
**(GIVE ME LINES CONTAINING COBALT)                                     [2]
SAMPLE   PHASE     CONSTIT.     CONTENT     UNIT     CITATION     TAG
S1ØØØ2   OVERALL   CO56         4Ø.Ø        DPM/KG   D7Ø-237      Ø
                   C13          8.8         DEL      D7Ø-228      Ø
                   H3           314.Ø       DPM/KG
                   MN54         28

**GETP(COBALT ALTFORMS)                                                 [3]
(CO56 CO57 CO6Ø C13 H3 MN54 NA22 SC46 S34 TI44)
**UNDO MAPCONC                                                          [4]
SETQ UNDONE.
**REDO GETP                                                             [5]
(CO56 CO57 CO6Ø)
**REDO COBALT                                                           [6]
SAMPLE   PHASE     CONSTIT.     CONTENT     UNIT     CITATION     TAG
S1ØØØ2   OVERALL   CO57         4Ø.Ø        DPM/KG   D7Ø-237      Ø
S1ØØØ3   OVERALL   CO           15.Ø                 D7Ø-2Ø3      Ø
                                14.1                 D7Ø-216
                   CO56         43.Ø        DPM/KG   D7Ø-237      Ø
                   CO57         43.Ø                 D7Ø-241      Ø
                   CO6Ø         1.Ø
**USE MANGANESE FOR COBALT
```

†The output is from the Lunar Sciences Natural Language Information System being developed for the NASA Manned Spacecraft Center by William A. Woods of Bolt Beranek and Newman Inc., Cambridge, Mass.

The user is running under his own executive program which accepts requests in the form of sentences, which it first parses and then executes. The user first 'innocently' computes a list of all ALTERNATIVE-FORMS for the elements in his system [1]. He then inputs a request in sentence format [2] expecting to see under the column CONSTIT. only cobalt, CO, or its alternate forms, CO56, CO57, or CO60. Seeing C13, H3, and MN54, he aborts the output, and checks the property ALTFORMS for COBALT [3]. The appearance of C13, H3, MN54 et al, remind him that the mapconc is destructive, and that in the process of making a list of the ALTFORMS, he has inadvertently strung them all together. Recovering from this situation would require him to individually examine and correct the ALTFORMs for each element in his dictionary, a tedious process. Instead, he can simply UNDO MAPCONC, [4] check to make sure the ALTFORM has been corrected [5], then redo his original request [6] and continue. The UNDO is possible because the first input was executed by lispx; the (GIVE ME LINES CONTAINING COBALT) is possible because the user defined userinputp appropriately; and the REDO and USE are possible because the (GIVE ME LINES CONTAINING COBALT) was stored on the history list before it was transmitted to lispxuserfn and the user's parsing program.

userinputp is a function of two arguments; let us call them x and y. When the user inputs in apply format, i.e. two expressions, userinputp is called with x the first expression and y the second. When the user inputs in eval format, i.e. one expression, userinputp is called with x=NIL, and y the expression. This permits distinguishing the case where the user types the single expression FOO, from the case of two expressions: FOO(). In the first case, x is NIL and y is FOO; in the second case, x is FOO and y is NIL.

Thus, in the above example, userinputp could be defined as (LAMBDA (X Y) (AND (NULL X) (LISTP Y))), and lispxuserfn would then be called to process all lists. Note that lispxuserfn could be programmed to call eval if the sentence did not parse correctly. Alternatively, the user could design his executive so that a list that 'looked like' a form was evaluated, not parsed, e.g. define userinputp as (LAMBDA (X Y) (AND (NULL X) (LISTP Y) (NOT (GETD (CAR Y].

lispxuserfn is a function of *three* arguments. The first two are the same as for userinputp, the third is the remainder of the input line. Note that userinputp is not given the rest of the line, since it is not known whether or not the rest of the line is supposed to be read.[†] Thus in the previous example, if the user wished all inputs typed in line format to be given to his parser, he could define userinputp as (LAMBDA (X Y) (AND X (NLISTP Y) (LISPXREADP))).[††] Then he could type GIVE ME LINES CONTAINING COBALT without parentheses.

⁂ ⁂ ⁂

In addition to the above features, lispx checks to see if car or cdr of NIL or car of T have been clobbered, and if so, restores them and prints a message. Lispx also performs spelling corrections using lispxcoms, a list of its commands, as a spelling list whenever it is given an unbound atom or undefined function, i.e. before attempting to evaluate the input.[†††]

[†]If userinputp itself reads the rest of the line, it must put the line back using lispxunread, described on page 22.50.

[††]lispxreadp is similar to readp but works when 'rereading.' If readp were used instead of lispxreadp, lispxuserfn would never be called for sentences being redone as a result of history commands.

[†††]lispx is also responsible for rebinding helpclock, used by breakcheck, p. 16.7, for computing the amount of time spent in a computation, in order to determine whether to go into a break if and when an error occurs.

Undoing

The UNDO capability of the programmer's assistant is implemented by requiring that each operation that is to be undoable be responsible itself for saving on the history list enough information to enable reversal of its side effects. In other words, the assistant does not 'know' when it is about to perform a destructive operation, i.e. it is not constantly checking or anticipating. Instead, it simply executes operations, and any undoable changes that occur are automatically saved on the history list by the responsible function.† The operation of UNDOing, which involves recovering the saved information and performing the corresponding inverses, works the same way, so that the user can UNDO an UNDO, and UNDO that etc.

At each point, until the user specifically requests an operation to be undone, the assistant does not know, or care, whether information has been saved to enable the undoing. Only when the user attempts to undo an operation does the assistant check to see whether any information has been saved. If none has been saved, and the user has specifically named the event he wants undone, the assistant types NOTHING SAVED. (When the user simply types UNDO, the assistant searches for the last undoable event, ignoring events already undone as well as UNDO operations themselves.)

†When the number of changes that have been saved exceeds the value of #undosaves (initially set to 50), the user is asked if he wants to continue saving the undo information for this event. The purpose of this feature is to avoid tying up large quantities of storage for operations that will never need be undone, e.g. loading a file. The interaction is handled by the same routines used by DWIM, so that the input buffers are first saved and cleared, the message typed, then the system waits dwimwait seconds, and if there is no response, assumes the default answer, which in this case is NO. Finally the input buffers are restored. See p. 22.56 for details.

This implementation minimizes the overhead for undoing. Only those operations which actually make changes are affected, and the overhead is small: two or three cells of storage for saving the information, and an extra function call. However, even this small price may be too expensive if the operation is sufficiently primitive and repetitive, i.e. the extra overhead may seriously degrade the overall performance of the program.† Hence not every destructive operation in a program should necessarily be undoable; the programmer must be allowed to decide each case individually.

Therefore for each primitive destructive operation, we have implemented *two* separate functions, one which always saves information, i.e. is always undoable, and one which does not, e.g. /rplaca and rplaca, /put and put.†† In the various system packages, the appropriate function is used. For example, break uses /putd and /remprop so as to be undoable, and DWIM uses /rplaca and /rplacd, when it makes a correction.††† Similarly the user can simply use the corresponding / function if he wants to make a destructive operation in his own program undoable. When the / function is called, it will save the undo information in the current event on the history list.

†The rest of the discussion applies only to lispx; the editor handles undoing itself in a slightly different fashion, as described on p. 22.59.

††The 'slash' functions currently implemented are /addprop, /attach, /dremove, /dreverse, /dsubst, /lconc, /mapcon, /mapconc, /movd, /nconc, /nconc1, /put, /putd, /putdq, /remprop, /rplaca, /rplacd, /set, /seta, and /tconc. Note that /setq and /setqq are *not* included. If the user wants a set operation undoable in his program, he must use /set.

†††The effects of the following functions are always undoable, regardless of whether or not they are typed in: define, defineq, defc (used to give a function a compiled code definition), deflist, load, savedef, unsavedef, break, unbreak, rebreak, trace, breakin, unbreakin, changename, editfns, editf, editv, editp, edite, editl, esubst, advise, unadvise, plus any changes caused by DWIM.

However, all operations that are *typed in* to lispx *are* made undoable, simply by substituting the corresponding / function for any destructive function throughout the input.† For example, on page 22.36, when the user typed (MAPCONC NASDIC (F/L ...)) it was (/MAPCONC NASDIC (F/L ...)) that was evaluated. Since the system cannot know whether efficiency and overhead are serious considerations for the execution of an expression in a user *program*, the user must decide, e.g. call /mapconc if he wants the operation undoable. However, expressions that are typed-in rarely involve iterations or lengthy computations *directly*. Therefore, if all primitive destructive functions that are immediately contained in a type-in are made undoable, there will rarely be a significant loss of efficiency. Thus lispx scans all user input before evaluating it, and substitutes the corresponding undoable function for all primitive destructive functions. Obviously with a more sophisticated analysis of both user input and user programs, the decision concerning which operations to make undoable could be better advised. However, we have found the configuration described here to be a very satisfactory one. The user pays a very small price for being able to undo what he types in, and if he wishes to protect himself from malfunctioning in his own programs, he can have his program specifically call undoable functions, or go into testmode as described next.

†Except when the input is a define (or defineq, putd, or putdq), the function definition is not touched. Similarly, on calls to break, getd, etc. references to destructive functions will not be changed to the corresponding / functions.

Testmode

Because of efficiency considerations, the user may not want certain functions undoable after his program becomes operational. However, while debugging he may find it desirable to protect himself against a program running wild, by making all primitive destructive operations undoable. The function testmode provides this capability by temporarily making everything undoable.

testmode[flg]	testmode[T] redefines all primitive destructive functions[†] with their corresponding undoable versions and sets testmodeflg to T. testmode[] restores the original definitions, and sets testmodeflg to NIL.[††]

[†] See second footnote on p. 22.40.

[††] testmode will have no effect on compiled mapconc's, since they compile open with frplacd's.

Undoing Out of Order

/rplaca and /rplacd operate by saving the pointer that is to be changed and its original contents (i.e. /rplaca saves car and /rplacd saves cdr). Undoing /rplaca and /rplacd simply restores the pointer. Thus, if the user types (RPLACA FOO 1), followed by (RPLACA FOO 2), then undoes both events by undoing the most recent event first, then undoing the older event, FOO will be restored to its state before either rplaca operated. However if the user undoes the first event, *then* the second event, (CAR FOO) will be 1, since this is what was in car of FOO before (RPLACA FOO 2) was executed. Similarly, if the user performs (NCONC1 FOO 1) then (NCONC1 FOO 2), undoing just (NCONC1 FOO 1) will remove both 1 and 2 from FOO. The problem in both cases is that the two operations are not 'independent.' In general, operations are always independent if they affect different lists or different sublists of the same list.† Undoing in reverse order of execution, or undoing independent operations, is always guaranteed to do the 'right' thing. However, undoing dependent operations out of order may not always have the predicted effect.

† Property list operations, (i.e. put, addprop and remprop) are handled specially so that they are always independent, even when they affect the same property list. For example, if the user types PUT(FOO FIE1 FUM1) then PUT(FOO FIE2 FUM2), then undoes the first event, the FIE2 property will remain, even though CDR(FOO) may have been NIL at the time the first event was executed.

Saveset

Setq's are made undoable on type in by substituting a call to saveset (described in detail on page 22.55), whenever setq is the name of the function to be applied, or car of the form to be evaluated.[†] In addition to saving enough information on the history list to enable undoing, saveset operates in a manner analogous to savedef when it resets a top level value, i.e. when it changes a top level binding from a value other than NOBIND to a new value that is not equal to the old one. In this case, saveset saves the old value of the variable being set on the variable's property list under the property VALUE, and prints the message (variable RESET). The old value can be restored via the function unset,[††] which also saves the current value (but does not print a message). Thus unset can be used to flip back and forth between two values.

rpaq and rpaqq are implemented via calls to saveset. Thus old values will be saved and messages printed for any variables that are reset as the result of loading a file.[†††] Calls to set and setqq appearing in type in are also converted to appropriate calls to saveset.

Saveset also adds its argument to the appropriate spelling list, thereby noticing variables set in files rpaq or rpaqq, as well as those set via type in.

[†]i.e. setq's are *not* undoable, even when typed in, if they appear somewhere *inside* of the expression, e.g. inside of a progn or lambda expression.

[††]Of course, UNDO can be used as long as the event containing this call to saveset is still active. Note however that the old value will remain on the property list, and therefore be recoverable via unset, even after the original event has been forgotten.

[†††]To complete the analogy with define, saveset will not save old values on property lists if dfnflg=T, e.g. when load is called with second argument T. However, the call to saveset will still be undoable.

Format and Use of the History List

There are currently two history lists, lispxhistory and edithistory. Both history lists have the same format, and in fact, each use the same function, historysave, for recording events, and the same set of functions for implementing commands that refer to the history list, e.g. historyfind, printhistory, undosave, etc.[†]

Each history list is a list of the form (l event# size mod), where l is the list of events with the most recent event first, event# is the event number for the most recent event on l, size is the size of the time-slice, i.e. the maximum length of l, and mod is the highest possible event number (see footnote on page 22.10). lispxhistory and edithistory are both initialized to (NIL 0 30 100). Setting lispxhistory or edithistory to NIL is permitted, and simply disables all history features, i.e. lispxhistory and edithistory act like flags as well as repositories of events.

Each individual event on l is a list of the form (input id value . props), where input is the input sequence for the event, as described on pp. 22.21-22, id the prompt character, e.g. ←, :, *,[††] and value is the value of the event, and is initialized to bell.[†††]

[†]A third history list, archivelst, is used when events are archived, as described on page 22.28. It too uses the same format.

[††]id is one of the arguments to lispx and to historysave. A user can call lispx giving it any prompt character he wishes (except for *, since in certain cases, lispx must use the value of id to tell whether or not it was called from the editor.) For example, on page 22.36, the user's prompt character was **.

[†††]On edithistory, this field is used to save the side effects of each command.

props is a property list, i.e. of the form (property value property value ...). props can be used to associate arbitrary information with a particular event. Currently, the properties SIDE, GROUP, HISTORY, PRINT, USE-ARGS, and ...ARGS, are being used. The value of property SIDE is a list of the side effects of the event. (See discussion of undosave and undolispx; pp. 22.56-57. The HISTORY and GROUP properties are used for commands that reexecute previous events, i.e. REDO, RETRY, USE, ..., and FIX. The value of the HISTORY property is the history command itself, that is what the user actually typed, e.g. REDO FROM F, and is used by the ?? command for printing the event. The value of the property PRINT is also for use by the ?? command, when special formatting is required, for example, in printing events corresponding to the break commands OK, GO, EVAL and ?=. USE-ARGS and ...ARGS are used to save the arguments and expression for the corresponding history command (see last paragraph, p. 22.18).

When lispx is given an input, it calls historysave to record the input in a new event.[†] Normally, historysave returns as its value cddr of the new event, i.e. car of its value is the value field of the event. lispx binds lispxhist to the value of historysave, so that when the operation has completed, lispx knows where to store the value, namely in car of lispxhist.[††] lispxhist also provides access to the property list for the current event. For example, the / functions are all implemented to call undosave, which simply adds the corresponding information to lispxhist under the property SIDE, or if there is no property SIDE, creates one, and then adds the information.

After binding lispxhist, lispx executes the input, stores its value in car of lispxhist, prints the value, and returns.

[†]The commands ??, FORGET, TYPE-AHEAD, $, and ARCHIVE are executed immediately, and are not recorded on the history list.

[††]Note that by the time it completes, the operation may no longer correspond to the most recent event on the history list. For example, all inputs typed to a lower break will appear later on the history list.

When the input is a REDO, RETRY, USE, ..., or FIX command, the procedure is similar, except that the event is also given a GROUP property, initially NIL, and a HISTORY property, and lispx simply unreads the input and returns. When the input is 'reread', it is historysave, not lispx, that notices this fact, and finds the event from which the input originally came.* historysave then adds a new (value . props) entry to the GROUP property for this event, and returns this entry as the 'new event.' lispx then proceeds exactly as when its input was typed directly, i.e. it binds lispxhist to the value of historysave, executes the input, stores the value in car of lispxhist, prints the value, and returns. In fact, lispx never notices whether it is working on freshly typed input, or input that was reread. Similarly, undosave will store undo information on lispxhist under the property SIDE the same as always, and does not know or care that lispxhist is not the entire event, but one of the elements of the GROUP property. Thus when the event is finished, its entry will look like:

```
(input id value HISTORY command GROUP ((value1 SIDE side1)
                                        (value2 SIDE side2)
                                               ...))†
```

This implementation removes the burden from the functions calling historysave of distinguishing between new input and reexecution of input whose history entry has already been set up.††

---

*If historysave cannot find the event, for example if a user program unreads the input directly, and not via a history command, historysave proceeds as though the input were typed.

†In this case, the value field is not being used; valueof instead collects each of the values from the GROUP property, i.e. returns mapcar[get[event;GROUP];CAR]. Similarly, undo operates by collecting the SIDE properties from each of the elements of the GROUP property, and then undoing them in reverse order.

††Although we have not yet done so, this implementation, i.e. keeping the various 'sub-events' separate with respect to values and properties, also permits constructing commands for operating on just one of the sub-events.

Functions

lispx[lispxx;lispxid][†]

lispx is like eval/apply. It carries out a single computation, and returns its value. The first argument, lispxx, is the result of a single call to lispxread. lispx will perform any additional lispxread's that are necessary, e.g. reading the arguments when lispxx is the first expression of an apply input. lispx prints the value of the computation, as well as saving the input. and value on lispxhistory.[††]

If lispxx is a history command, lispx executes the command, and returns bell as its value.

The overhead for a call to lispx is approximately 17 milliseconds, of which 12 milliseconds are spent in saving the input on the history list, and 4 milliseconds in maintaining the spelling lists. In other words, in BBN-LISP, the user pays 17 more milliseconds for each eval or apply input over a conventional LISP executive, in order to enable the features described in this chapter.

[†] Lispx has a third argument, lispxflg, which is used by the E command in the editor.

[††] Note that the history list is *not* one of the arguments to lispx, i.e. the editor must bind lispxhistory to edithistory before calling lispx to carry out a history command.

Lispx will continue to operate as an eval/apply function if lispxhistory is NIL. Only those functions and commands that involve the history list will be affected.

lispxread[file] is a generalized <u>read</u>. If <u>readbuf</u>=NIL, <u>lispxread</u> performs read[file], which it returns as its value. (If the user types control-U during the call to <u>read</u>, <u>lispxread</u> calls the editor and returns the edited value.)

If <u>readbuf</u> is not NIL, <u>lispxread</u> 'reads' the next expression on <u>readbuf</u>, i.e. essentially returns

```
(PROG1 (CAR READBUF)
       (SETQ READBUF (CDR READBUF))).†
```

<u>readline</u>, described on p. 14.11, also uses this generalized notion of reading. When <u>readbuf</u> is not NIL, <u>readline</u> 'reads' expressions from <u>readbuf</u> until it either reaches the end of <u>readbuf</u>, or until it reads a (VAG Ø). In both cases, it returns a list of the expressions it has 'read'. (The (VAG Ø) is not included in the list.)

†Except that pseudo-carriage returns, as represented by (VAG Ø), are ignored, i.e. skipped. <u>Lispxread</u> also sets <u>rereadflg</u> to NIL when it reads via <u>read</u>, and sets <u>rereadflg</u> to <u>readbuf</u> when rereading.

lispxreadp[flg] is a generalized readp. If flg=T, lispxreadp returns T if there is any input waiting to be 'read', a la lispxread. If flg=NIL, lispxreadp returns T only if there is any **input** waiting to be 'read' *on this line*. The definition of lispxreadp is:

```
(LISPXREADP
  [LAMBDA (FLG)

          (* If FLG is T, acts like READP, otherwise like a
          READP which only looks at this line.)


    (COND
      [READBUF (OR FLG (NEQ (CAR READBUF)
                            (VAG Ø]
      ((READP T)
        (OR FLG (NEQ (PEEKC T)
                     (QUOTE %
 ])
```

lispxunread[lst;copyflg] unreads lst, a list of expressions to be read. If readbuf is not NIL, lispxunread attaches lst at the front of readbuf, separating it from the rest of readbuf with a (VAG Ø). The definition of lispxunread is:

```
(LISPXUNREAD
  [LAMBDA (L COPYFLG)
    (SETQ READBUF (COND
        ((NULL READBUF)
          L)
        (COPYFLG (APPEND L (CONS (VAG Ø)
                                 READBUF)))
        (T (NCONC L (CONS (VAG Ø)
                          READBUF])
```

promptchar[id;flg;hist]   prints the prompt character id.

promptchar will not print anything when the next input will be 'reread', i.e. readbuf is not NIL. promptchar will also not print when readp[]=T, unless flg is T.

Thus the editor calls promptchar with flg=NIL so that extra *'s are not printed when the user types several commands on one line. However, evalqt calls promptchar with flg=T since it always wants the ← printed (except when 'rereading').

Finally, if prompt#flg is T and hist is not NIL, promptchar prints the current event number (of hist) before printing id.

lispxeval[lispxform;lispxid]   evaluates lispxform (using eval), the same as though it were typed in to lispx, i.e. the event is recorded, and the evaluation is made undoable by substituting the slash functions for the corresponding destructive functions as described on p. 22.41. lispxeval returns the value of the form but does not print it.

historysave[history;id;input1;input2;input3;props]

records one event on history. If input1 is not NIL, the input is of the form (input1 input2 . input3); historysave is called this way for recording apply inputs. If input1 is NIL, and input2 is not NIL, the input is of the form (input2 . input3); historysave is called this way for recording eval inputs. Otherwise, the input is just input3; historysave is called this way for recording line commands.†

historysave creates a new event with the corresponding input, id, value field initialized to bell, and props. If the history has reached its full size, the last event is removed and cannibalized.

The value of historysave is cddr of the event. However, if rereadflg is not NIL, and is a tail of the input of the most recent event on the history list, and this event contains a GROUP property, historysave does not create a new event, but simply adds a (bell . props) entry to the GROUP property and returns that entry. See discussion on p. 22.47.

†The reason for the three methods of specifying the input is to enable more complete reutilization of storage. Thus, lispx could call historysave with input1=NIL, input2=NIL, and input3=(GETD (FOO)), and get an entry equivalent to calling historysave with input1=GETD, input2=(FOO), input3=NIL. However, the latter avoids making up the list (GETD (FOO)) by reusing the input list of the event that is about to be forgotten.

lispxfind[history;line;type;backup]

line is an event specification, type specifies the format of the value to be returned by lispxfind, and can be either ENTRY, ENTRIES, COPY, COPIES, INPUT, or REDO. lispxfind parses line, and uses historyfind to find the corresponding events. lispxfind then assembles and returns the appropriate structure.

lispxfind incorporates the following special features:

(1) if backup=T, lispxfind interprets line in the context of the history list *before* the current event was added. This feature is used, for example, by valueof, so that (VALUEOF -1) will not refer to the valueof event itself;

(2) if line=NIL and the last event is an UNDO, the next to the last event is taken. This permits the user to type UNDO followed by REDO or USE;

(3) lispxfind recognizes @@, and substitutes archivelst for history (see p. 22.17); and

(4) lispxfind recognizes @, and retrieves the corresponding event(s) from the property list of the atom following @ (see p. 22.17).

historyfind[lst;index;mod;x;y]

searches lst and returns the tails of lst beginning with the event corresponding to x. lst, index, and mod are as described on p. 22.45.

x is an event address, as described on pp. 22.15-16, e.g. (43), (-1), (FOO FIE) (LOAD ← FOO), etc.[†] If historyfind cannot find x, it generates an error.

[†]If y is given, the event address is the *list difference* between x and y, e.g. x=(FOO FIE AND \ -1), y=(AND \ -1) is equivalent to x=(FOO FIE), y=NIL.

entry#[hist;x] hist is a history list, i.e. of the form described on p. 22.45. X is one of the events on hist, i.e. (MEMB X (CAR HIST)) is true. The value of entry# is the event number for x.

valueof[x] is an nlambda, nospread function for obtaining the value of the event specified by x, e.g. (VALUEOF -1), (VALUEOF LOAD 1), etc. valueof returns a list of the corresponding values if x specifies an event with a GROUP property, i.e. an event corresponding to a REDO, RETRY, USE, ..., or FIX command.

changeslice[n;history] changes time slice for history to n. If history is NIL, changes both edithistory and lispxhistory.

Note: the effect of *increasing* a time-slice is gradual: the history list is simply allowed to grow to the corresponding length before any events are forgotten. *Decreasing* a time-slice will immediately remove a sufficient number of the older events to bring the history list down to the proper size. However, changeslice is undoable, so that these events are (temporarily) recoverable. Thus if the user wants to recover the storage associated with these events without waiting n more events for the changeslice event to be forgotten, he must perform a FORGET -1 command.

saveset[name;value;topflg;flg]

an undoable set. (see p. 22.44). saveset scans the pushdown list looking for the last binding of name, sets name to value, and returns value.

If the binding changed was a top level binding, and the old value was not NOBIND and was not equal to value, saveset prints (name RESET), and saves the old value on the property list of name, under the property VALUE.

If topflg=T, saveset does not scan the pushdown list but goes right to name's value cell, e.g. rpaqq[x;y] is simply saveset[x;y;T].

If flg=NOSAVE, or dfnflg=T, saveset will not save the old value on the property list, e.g. /set[x;y] is saveset[x;y;NIL;NOSAVE].

If flg=NOPRINT, saveset does not print any message; unset uses this option.

unset[name]

if name does not contain a property VALUE, unset generates an error. Otherwise unset calls saveset with name, the property value, topflg=T, and flg=NOPRINT.

undosave[x]

if lispxhist is not NIL[†] (see discussion on p. 22.46), undosave adds x to the value of the property SIDE on lispxhist, creating a SIDE property if one does not already exist. The form of x is (fn . args), i.e. x is undone by performing apply[car[x];cdr[x]]. For example, if y=(A B C), (/RPLACA Y (QUOTE D)) will call undosave with x=(/RPLACA (D B C) A).

car of the SIDE property is the number of 'undosaves', i.e. length of cdr of the SIDE property. Each call to undosave increments this count, and adds x to the front of the list, i.e. just after the count. When the count reaches the value of #undosaves (initially 50), undosave prints a message asking the user if he wants to continue saving. If the user answers NO or defaults, undosave sets lispxhist to NIL which disables any further saving for this event. If the user answers YES, undosave changes the count to -1, which is then never incremented, and continues saving.

new/fn[fn]

After the user has defined /fn, new/fn performs the necessary housekeeping operations to make fn be undoable.

For example, the user could define /radix as (LAMBDA (X) (UNDOSAVE (LIST (QUOTE /RADIX) (RADIX X)))) and then perform new/fn[radix], and radix would then be undoable.

[†] If lispxhist = NIL, undosave is a NOP.

undolispx[line]	line is an event specification. undolispx is the function that executes UNDO commands by calling undolispxl on the appropriate entry(s).
undolispxl[event;flg]	undoes one event. The value of undolispxl is NIL if there is nothing to be undone. If the event is already undone, undolispxl prints ALREADY UNDONE and returns T.† Otherwise, undolispxl undoes the event, prints a message, e.g. SETQ UNDONE, and returns T.

Undoing an event consists of mapping down (cdr of) the property value for SIDE, and for each element, applying car to cdr, and then marking the event undone by attaching (with /attach) a NIL to the front of its SIDE property. Note that the undoing of each element on the SIDE property will usually cause undosaves to be added to the *current* lispxhist, thereby enabling the effects of undolispxl to be undone.

† If flg=T and the event is already undone, or is an undo command, undolispxl takes no action and returns NIL. Undolispx uses this option to search for the last event to undo. Thus when line=NIL, undolispx simply searches history until it finds an event for which undolispxl returns T, i.e. undolispx performs (SOME (CDAR LISPXHISTORY) (F/L (UNDOLISPXl X T)))

printhistory[history;line;skipfn;novalues]

line is an event specification. printhistory prints the events on history specified by line. skipfn is an (optional) functional argument that is applied to each event. If its value is true, the event is skipped, i.e. not printed. If novalues = T, or novalues applied to the corresponding event is true, the value is not printed.[†]

For example, the following lispxmacro will define ??' as a command for printing the history list while skipping all 'large events' and not printing any values.

```
(??' (PRINTHISTORY LISPXHISTORY LISPXLINE
          (FUNCTION (LAMBDA (X)
             (IGREATERP (COUNT (CAR X)) 5)))
         T))
```

[†] novalues is automatically T for printing edithistory.

The Editor and the Assistant

As mentioned earlier, all of the remarks concerning 'the assistant' apply equally well to user interactions with evalqt, break or the editor. The differences between the editor's implementation of these features and that of lispx are mostly obvious or inconsequential. However, for completeness, this section discusses the editor's implementation of the programmer's assistant.

The editor uses promptchar to print its prompt character, and lispxread, lispxreadp, and readline for obtaining inputs. When the editor is given an input, it calls historysave to record the input in a new event on its history list, edithistory.† Edithistory follows the same conventions and format as lispxhistory. However, since edit commands have no value, the editor uses the value field for saving side effects, rather than storing them under the property SIDE.

The editor processes DO, !E, !F, and !N commands itself, since lispx does not recognize these commands. The editor also processes UNDO itself, as described below. All other history commands†† are simply given to lispx for execution, after first binding lispxhistory to edithistory. The editor also calls lispx when give an E command as described on p. 9.69.†††

† Except that the atomic commands OK, STOP, SAVE, P, ?, PP, and E are not recorded. In addition, number commands are grouped together in a single event. For example, 3 3 -1 is considered as one command for changing position.

†† as indicated by their appearance on historycoms, a list of the history commands. editdefault interrogates historycoms before attempting spelling correction. (All of the commands on historycoms are also on editcomsa and editcomsl so that they can be corrected if misspelled in the editor.) Thus if the user defines a lispxmacro and wishes it to operate in the editor as well, he need simply add it to historycoms. For example, RETRIEVE is implemented as a lispxmacro and works equally well in lispx and the editor.

††† In this case, the editor uses the third argument to lispx, lispxflg, to specify that all history commands are to be executed by a recursive call to lispx, rather than by unreading. For example, if the user types E REDO in the editor, he wants the last event on lispxhistory processed as lispx input, and not to be unread and processed by the editor.

The major implementation difference between the editor and lispx occurs in undoing. Edithistory is a list of only the last n commands,where n is the value of the time-slice. However the editor provides for undoing *all* changes made in a single editing session, even if that session consisted of more than n edit commands. Therefore, the editor saves undo information independently of the edithistory on a list call undolst, (although it also stores each entry on undolst in the value field of the corresponding event on edithistory.) Thus, the commands UNDO, !UNDO, and UNBLOCK, are not dependent on edithistory,[†] i.e. UNDO specifies undoing the last command on undolst, even if that event no longer appears on edithistory. The only interaction between UNDO and the history list occurs when the user types UNDO followed by an event specification. In this case, the editor calls lispxfind to find the event, and then undoes the corresponding entry on undolst. Thus the user can only undo a *specified* command within the scope of the edithistory. (Note that this is also the only way UNDO commands themselves can be undone, that is, by using the history feature, to specify the corresponding event, e.g. UNDO UNDO.)

The implementation of the undoing itself is similar to the way it is done in lispx: each command that makes a change in the structure being edited does so via a function that records the change on a variable. After the command has completed, this variable contains a list of all the pointers that have been changed and their original contents. Undoing that command simply involves mapping down that list and restoring the pointers.

[†]and in fact will work if edithistory=NIL, or even in a system which does not contain lispx at all.

APPENDICES

APPENDIX 1 - TRANSOR

Contents

Introduction

TRANSOR is a LISP-to-LISP translator intended to help the user who has a program coded in one dialect of LISP and wishes to carry it over to another. The user loads TRANSOR along with a file of transformations. These transformations describe the differences between the two LISPs, expressed in terms of the BBN editor commands needed to convert the old to new, i.e. to edit forms written in the source dialect to make them suitable for the target dialect. TRANSOR then sweeps through the user's program and applies the edit transformations, producing an object file for the target system. In addition, TRANSOR produces a file of translation notes, which catalogs the major changes made in the code as well as the forms that require further attention by the user. Operationally, therefore, TRANSOR is a facility for conducting massive edits, and may be used for any purpose which that may suggest.

Since the edit transformations are fundamental to this process, let us begin with a definition and some examples. A transformation is a list of edit commands associated with a literal atom, usually a function name. TRANSOR conducts a sweep through the user's code, until it finds a form whose car is a literal atom which has a transformation. The sweep then pauses to let the editor execute the list of commands before going on. For example, suppose the order of arguments for the function tconc must be reversed for the target system. The transformation for tconc would then be: ((SW 2 3)). When the sweep encounters the form (TCONC X (FOO)), this transformation would be retrieved and executed, converting the expression to (TCONC (FOO) X). Then the sweep would locate the next form, in this case (FOO), and any transformations for foo would be executed, etc.

Most instances of tconc would be successfully translated by this transformation. However, if there were no second argument to tconc, e.g. the form to be translated was (TCONC X), the command (SW 2 3) would cause an error, which TRANSOR would catch. The sweep would go on as before, but a note would appear in the translation listing stating that the transformation for this particular form failed to work. The user would then have to compare the form and the commands, to figure out what caused the problem. One might, however, anticipate this difficulty with a more sophisticated transformation: ((IF (## 3) ((SW 2 3)) ((-2 NIL)))), which tests for a third element and does (SW 2 3) or (-2 NIL) as appropriate. It should be obvious that the translation process is no more sophisticated than the transformations used.

This documentation is divided into two main parts. The first describes how to use TRANSOR assuming that the user already has a complete set of transformations. The second documents TRANSORSET, an interactive routine for building up such sets. TRANSORSET contains commands for writing and editing transformations, saving one's work on a file, testing transformations by translating sample forms, etc.

Neither TRANSOR nor TRANSORSET are included in the regular BBN LISP system. To get TRANSOR, load <LISP>TRANSOR.COM. To get TRANSORSET, load <LISP>TSET.COM.†

Two transformations files presently exist for translating programs into the current BBN LISP. <LISP>SDS940.XFORMS is for old BBN LISP (SDS 940) programs, and <LISP>LISP16.XFORMS is for Stanford AI LISP 1.6 programs. A set for LISP 1.5 is planned.

Using TRANSOR

The first and most exasperating problem in carrying a program from one implementation to another is simply to get it to read in. For example, SRI LISP processes tabs specially for formatting, but tabs are not special characters to BBN LISP, and so they read in as parts of adjacent atoms, prog labels, etc. Also, SRI LISP uses / exactly as BBN uses %, i.e. as an escape character. The function prescan exists to help with these problems: the user uses prescan to perform an initial scan to dispose of these difficulties, rather than attempting to TRANSOR the foreign sourcefiles directly.

prescan copies a file, performing character-for-character substitutions. It is hand-coded and is much faster than either readc's or text-editors.

prescan[file;charlst]	Makes a new version of file, performing substitutions according to charlst. Each element of charlst must be a dot-pair of two character codes, (OLD . NEW).

†The interpreted code is on <LISP>TRANSOR and <LISP>TSET, and is copiously commented.

For example, SRI files are prescan'ed with charlst = ((9 · 32) (37 · 47) (47 · 37)). This converts tabs (9) to spaces (32), and exchanges slash (47) and percent-sign (37).

The user should also make sure that the treatement of double-quotes by the source and target systems is similar. In BBN LISP an unmatched double-quote (unless protected by the escape character) will cause the rest of the file to read in as a string.[†]

Finally, the lack of a STOP at the end of a file is harmless, since TRANSOR will suppress END OF FILE errors and exit normally.

Translating

TRANSOR is the top-level function of the translator itself, and takes one argument, a file to be translated. The file is assumed to contain a sequence of forms, which are read in, translated, and output to a file called file.TRAN. The translation notes are meanwhile output to file.LSTRAN. Thus the usual sequence for bringing a foreign file to BBN is as follows: prescan the file; examine code and transformations, making changes to the transformations if needed; transor the file; list both output and translation notes; load the output file; and clean up remaining problems, guided by the notes. The user can now make a pretty file and proceed to exercise and check out his program. To export a file, it is usually best to transor it, then prescan it, and perform clean-up on the foreign system where the file can be loaded.

[†] A special case of this is the problem of bringing programs from old BBN LISP (before strings were implemented) to the latest version. A special interim READ was implemented for this task which is available from archives.

transor[sourcefile]	Translates sourcefile. Prettyprinted translation on file.TRAN; translation listing on file.LSTRAN.
transorform[form]	Argument is a LISP form. Returns the (destructively) translated form. The translation listing is dumped to the primary output file.
transorfns[fnlst]	Argument is a list of function names whose interpreted definitions are destructively translated. Listing to primary output file.

transorform and transorfns can be used to translate expressions that are already in core, whereas transor itself only works on files.

The Translation Notes

That translation of notes are a catalog of changes made in the user's code, and of problems which require, or may require, further attention from the user. This catalog consists of two cross-indexed sections: an index of forms and an index of notes. The first tabulates all the notes applicable to any one form, whereas the second tabulates all the forms to which any one note applies. Forms appear in the index of forms in the order in which they were encountered, i.e. the order in which they appear on the source and output files. The index of notes shows the name of each note, the entry numbers where it was used, and its text, and is alphabetical by name. The following sample was made by translating a small test file written in SRI LISP.

```
      LISTING FROM TRANSORING OF FILE TESTFILE.;5
      DONE ON   1-NOV-71 2Ø:1Ø:47

                          INDEX OF FORMS


1. APPLY/EVAL at
  [DEFINEQ (FSET (LAMBDA &
                   (PROG ...3...
                         (SETQ Z (COND
                             ((ATOM (SETQ --))
                               (COND
                                 ((ATOM (SETQ Y (NLSETQ "(EVAL W)")))
                                   --)
                                 --))
                             --))
                     --  ]
2. APPLY/EVAL at
  [DEFINEQ (FSET (LAMBDA &
                   (PROG ...3...
                         (SETQ Z (COND
                             ((ATOM (SETQ --))
                               (COND
                                 ((ATOM (SETQ --))
                                   "(EVAL (NCONS W))")
                                 --))
                             --))
                     --  ]
3. MACHINE-CODE at
  [DEFINEQ (LESS1 (LAMBDA &
                    (PROG ...3...
                          (COND
                            ...2...

                            ((NOT (EQUAL (SETQ X2 "(OPENR (MAKNUM & --))"
                                               )
                                             --))
                              --))
                      --  ]
4. MACHINE-CODE at
  [DEFINEQ (LESS1 (LAMBDA &
                    (PROG ...3...
                          (COND
                            ...2...
                            ((NOT (EQUAL & (SETQ Y2
                                               "(OPENR (MAKNUM & --))")))
                              --))
                      --  ]
```

```
                         INDEX OF NOTES
APPLY/EVAL at 1, 2.
     TRANSOR will translate the arguments of the APPLY or EVAL
expression, but the user must make sure that the run-time evaluation of
the arguments returns a BBN-compatible expression.
MACHINE-CODE at 3, 4.
     Expression dependent on machine-code. User must recode.
```

The translation notes are generated by the transformations used, and therefore reflect the judgment of their author as to what should be included. Straightforward conversions are usually made without comment; for example, the DEFPROP's in this file were quietly changed to DEFINEQ's. TRANSOR found four noteworthy forms on the file, and printed an entry for each in the index of forms, consisting of an entry number, the name of the note, and a printout showing the precise location of the form. The form appears in double-quotes and is the last thing printed, except for closing parentheses and dashes. An ampersand represents one non-atomic element not shown, and two or more elements not shown, and two or more elements not shown are represented as ...n... where n is the number of elements. Note that the printouts describe expressions on the output file rather than the source file; in the example, the DEFPROP's of SRI LISP have been replaced with DEFINEQ's.

Errors and Messages

TRANSOR records its progress through the source file by teletype printouts which identify each expression as it is read in. Progress within large expressions, such as a long DEFINEQ, is reported every three minutes by a printout showing the location of the sweep.

If a transformation fails, TRANSOR prints a diagnostic to the teletype which identifies the faulty transformation, and resumes the sweep with the next form. The translation notes will identify the form which caused this failure, and the extent to which the form and its arguments were compromised by the error.

If the transformation for a common function fails repeatedly, the user can type control-H. When the system breaks, he can use transorset to repair the transformation, and even test it out (see TEST command, p. A1.12). He may then continue the main translation with OK.

TRANSORSET

To use transorset, type TRANSORSET() to LISP. transorset will respond with a + sign, its prompt character, and await input. The user is now in an executive loop which is like evalqt with some extra context and capabilities intended to facilitate the writing of transformations. transorset will thus process apply and eval input, and execute history commands just as evalqt would. Edit commands, however, are interpreted as additions to the transformation on which the user is currently working. transorset always saves on a variable named currentfn the name of the last function whose transformation was altered or examined by the user. currentfn thus represents the function whose transformation is currently being worked on. Whenever edit commands are typed to the + sign, transorset will add them to the transformation for currentfn. This is the basic mechanism for writing a transformation. In addition, transorset contains commands for printing out a transformation, editing a transformation, etc., which all assume that the command applies to currentfn if no function is specified. The following example illustrates this process.

```
←TRANSORSET()
+FN TCONC                                                   [1]
TCONC
+(SW 2 3)                                                   [2]
+TEST (TCONC A B)                                           [3]
P
(TCONC B A)
+TEST (TCONC X)                                             [4]
TRANSLATION ERROR: FAULTY TRANSFORMATION
TRANSFORMATION: ((SW 2 3))                                  [5]
OBJECT FORM:     (TCONC X)


1. TRANSFORMATION ERROR AT                                  [6]
  "(TCONC X)"


(TCONC X)
+(IF(## 3)((SW 2 3))((-2 NIL]                               [7]
+SHOW
TCONC
  [(SW 2 3)
   (IF (## 3)                                               [8]
       ((SW 2 3))
       ((-2 NIL]
TCONC
+ERASE                                                      [9]
TCONC
+REDO IF                                                   [10]
+SHOW
TCONC
  [(IF (## 3)
       ((SW 2 3))
       ((-2 NIL]
TCONC
+TDST
=TEST                                                      [11]
(TCONC NIL X)
+
```

In this example, the user begins by using the FN command to set <u>currentfn</u> to TCONC [1]. He then adds to the (empty) transformation for <u>tconc</u> a command to switch the order of the arguments [2] and tests the transformation [3]. His second TEST [4] fails, causing an error diagnostic [5] and a translation note [6]. He writes a better command [7] but forgets that the SW command is still in the way [8]. He therefore deletes the entire transformation [9] and redoes the IF [10]. This time, the TEST works [11].

TRANSORSET Commands

The following commands for manipulating transformations are all <u>lispxmacros</u> which treat the rest of their input line as arguments. All are undoable.

FN	Resets <u>currentfn</u> to its argument, and returns the new value. In effect FN says you are done with the old function (at least for the moment) and wish to work on another. If the new function already has a transformation, the message (OLD TRANSFORMATIONS) is printed, and any editcommands typed in will be added to the end of the existing commands. FN followed by a carriage return will return the value of <u>currentfn</u> without changing it.

SHOW	Command to prettyprint a transformation. SHOW followed by a carriage return will show the transformation for *currentfn*, and return *currentfn* as its value. SHOW followed by one or more function names will show each one in turn, reset *currentfn* to the last one, and return the new value of *currentfn*.
EDIT	Command to edit a transformation. Similar to SHOW except that instead of prettyprinting the transformation, EDIT gives it to *edite*. The user can then work on the transformation until he leaves the editor with OK.
ERASE	Command to delete a transformation. Otherwise similar to SHOW.
TEST	Command for checking out transformations. TEST takes one argument, a form for translation. The translation notes, if any, are printed to the teletype, but in an abbreviated format which omits the index of notes. The value returned is the translated form. TEST saves a copy of its argument on the free variable *testform*, and if no argument is given, it uses *testform*, i.e. tries the previous test again.

DUMP	Command to save your work on a file. DUMP takes one argument, a filename. The argument is saved on the variable dumpfile, so that if no argument is provided, a new version of the previous file will be created.

The DUMP command creates files by makefile. Normally fileFNS will be unbound, but the user may set it himself; functions called from a transformation by the E command may be saved in this way. DUMP makes sure that the necessary command is included on the fileVARS to save the user's transformations. The user may add anything else to his fileVARS that he wishes. When a transformation file is loaded, all previous transformations are erased unless the variable merge is set to T.

EXIT	TRANSORSET returns NIL.

The REMARK Feature

The translation notes are generated by those transformations that are actually executed via an editmacro called REMARK. REMARK takes one argument, the name of a note. When the macro is executed, it saves the appropriate information for the translation notes, and adds one entry to the index of forms. The location that is printed in the index of forms is the editor's location when the REMARK macro is executed.

To write a transformation which makes a new note, one must therefore do two things: define the note, i.e. choose a new name and associate it with the desired text; and call the new note with the REMARK macro, i.e. insert the edit command (REMARK name) in some transformation. The NOTE command, described below, is used to define a new note. The call to the note may be added to a transformation like any other edit command. Once a note is defined, it may be called from as many different transformations as desired.

The user can also specify a remark with a new text, without bothering to think of a name and perform a separate defining operation, by calling REMARK with more than one argument, e.g. (REMARK text of remark). This is interpreted to mean that the arguments are the text. TRANSORSET notices all such expressions as they are typed in, and handles naming automatically: a new name is generated[†] and defined with the text provided, and the expression itself is edited to be (REMARK generated-name). The following example illustrates the use of REMARK.

[†]The name generated is the value of <u>currentfn</u> suffixed with a colon, or with a number and a colon.

```
←TRANSORSET()
+NOTE GREATERP/LESSP (BBN'S GREATERP AND LESSP ONLY          [1]
TAKE TWO ARGUMENTS, WHEREAS SRI'S FUNCTIONS TAKE AN
INDEFINITE NUMBER. AT THE PLACES NOTED HERE, THE SRI CODE
USED MORE THAN TWO ARGUMENTS, AND THE USER MUST RECODE.]
GREATERP/LESSP
+FN GREATERP                                                 [2]
GREATERP
+(IF(IGREATERP(LENGTH(##))3)NIL((REMARK GREATERP/LESSP]
+FN LESSP
LESSP
+REDO IF                                                     [3]
+SHOW
LESSP
  [(IF (IGREATERP (LENGTH (##))
                  3)
       NIL
       ((REMARK GREATERP/LESSP]
LESSP
+FN ASCII
(OLD TRANSFORMATIONS)
ASCII
+(REMARK ALTHOUGH THE SRI FUNCTION ASCII IS IDENTICAL        [4]
TO THE BBN FUNCTION CHARACTER, THE USER MUST MAKE SURE
THAT THE CHARACTER BEING CREATED SERVES THE SAME PURPOSE
ON BOTH SYSTEMS, SINCE THE CONTROL CHARACTERS ARE
ALL ASSIGNED DIFFRNTLY.]
+SHOW                                                        [5]
ASCII
  ((1 CHARACTER)
   (REMARK ASCII:))
ASCII
+NOTE ASCII:                                                 [6]
EDIT
*NTH -2
*P
... ASSIGNED DIFFRNTLY.)
*(2 DIFFERENTLY.)
*OK
ASCII:
+
```

In this example, the user defines a note named GREATERP/LESSP by using the NOTE command [1], and writes transformations which call this note whenever the sweep encounters a GREATERP or LESSP with more than two arguments [2-3]. Next, the implicit naming feature is used [4] to add a REMARK command to the transformation for ASCII, which has already been partly written. The user realizes he mistyped part of the text, so he uses the SHOW command to find the name chosen for the note [5]. Then he uses the NOTE command on this name,ASCII:, to edit the note [6].

NOTE	First argument is note name and must be a literal atom. If already defined, NOTE edits the old text; otherwise it defines the name, reading the text either from the rest of the input line or from the next line. The text may be given as a line or as a list. Value is name of note.

The text is actually stored[†] as a comment, i.e. a * and %% are added in front when the note is first defined. The text will therefore be lower-cased the first time the user DUMPs (see pp. 14.35-38).

DELNOTE	Deletes a note completely (although any calls to it remain in the transformations).

[†]on the global list usernotes.

Controlling the Sweep

TRANSOR's sweep searches in print-order until it finds a form for which a transformation exists. The location is marked, and the transformation is executed. The sweep then takes over again, beginning from the marked location, no matter where the last command of the transformation left the editor. User transformations can therefore move around freely to examine the context, without worrying about confusing the translator. However, there are many cases where the user wants his transformation to guide the sweep, usually in order to direct the processing of special forms and FEXPR's. For example, the transformation for QUOTE has only one objective: to tell the sweep to skip over the argument to QUOTE, which is (presumably) not a LISP form. NLAM is an editmacro to permit this.

NLAM	An atomic editmacro which sets a flag which causes the sweep to skip the arguments of the current form when the sweep resumes.

Special forms such as cond, prog, selectq, etc., present a more difficult problem. For example, (COND (A B)) is processed just like (FOO (A B)): i.e. after the transformation for cond finishes, the sweep will locate the "next form," (A B), retrieve the transformation for the function A, if any, and execute it. Therefore, special forms must have transformations that preempt the sweep and direct the translation themselves. The following two atomic editmacros permit such transformations to process their forms, translating or skipping over arbitrary subexpressions as desired.

DOTHIS	Translates the editor's current expression, treating it as a single form.

DOTHESE	Translates the editor's current expression, treating it as a list of forms.

For example, a transformation for <u>setq</u> might be (3 DOTHIS).[†] This translates the second argument to a <u>setq</u> without translating the first, which is quoted. For <u>cond</u>, one might write (1 (LPQ NX DOTHESE)), which locates each clause of the COND in turn, and translates it as a list of forms, instead of as a single form.

The user who is starting a completely new set of transformations must begin by writing transformations for all the special forms. To assist him in this and prevent oversights, the file <LISP>SPECIAL.XFORMS contains a set of transformations for LISP special forms, as well as some other transformations which should also be included. The user will probably have to revise these transformations substantially, since they merely perform sweep control for BBN LISP, i.e. they make no changes in the object code. They are provided chiefly as a checklist and tutorial device, since these transformations are both the first to be written and the most difficult, especially for users new to the BBN editor.

[†] Recall that a transformation is a *list* of edit commands. In this case, there are two commands, 3 and DOTHIS.

* * *

When the sweep mechanism encounters a form which is not a list, or a form car of which is not an atom, it retrieves one of the following special transformations.

NLISTPCOMS	Global value is used as a transformation for any form which is not a list.

For example, if the user wished to make sure that all strings were quoted, he might set nlistpcoms to
((IF (STRINGP(##)) ((ORR ((← QUOTE))((MBD QUOTE)))) NIL)).

LAMBDACOMS	Global value is used as a transformation for any form, car of which is not an atom.

These variables are initialized by <LISP>SPECIAL.XFORMS and are saved by the DUMP command. NLISTPCOMS is initially NIL, making it a NOP. LAMBDACOMS is initialized to check first for open LAMBDA expressions, processing them without translation notes unless the expression is badly formed. Any other forms with a non-atomic car are simply treated as lists of forms and are always mentioned in the translation notes. The user can change or add to this algorithm simply by editing or resetting LAMBDACOMS.

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 pointer to a block of compiled code; or

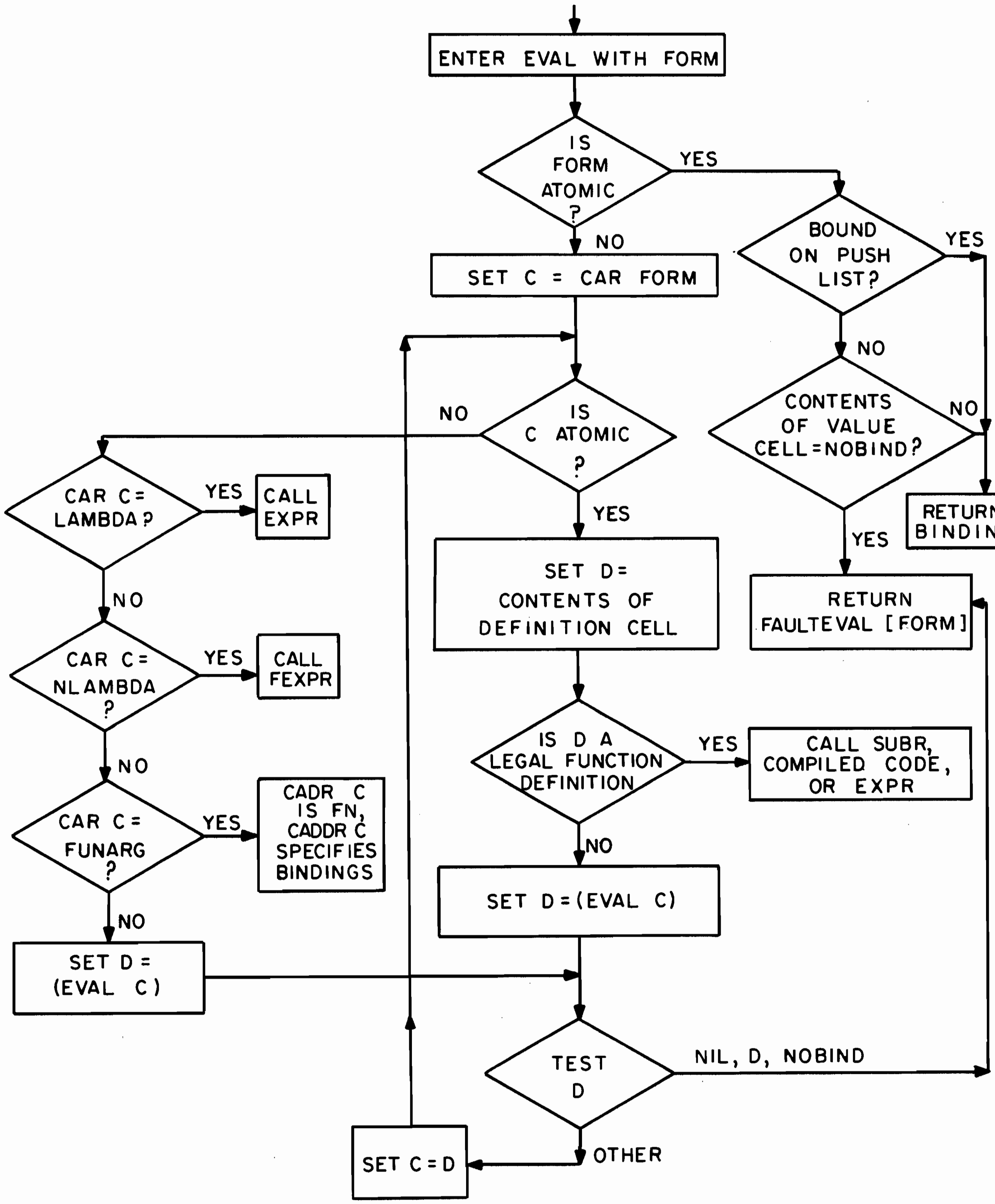
(c) a SUBR definition (see Section 8).

Otherwise, if the function cell contains NIL or an S-expression which is not a legal function definition, the atom is evaluated. (This is the way functional arguments work.) If the evaluation of the atom produces NIL or the atom itself, or if the atom is unbound, the form is considered faulty, and faulteval is called with the *original* form as its argument as described in Section 16. Otherwise the value of the atom is treated as if it were car of the original form, i.e., if the value is an atom, its function cell is consulted; if the value is an S-expression, proceed as described below. In other words, it is possible to bind an atom to either the *name* of a function, or a form which *computes* the name of a function, and then to use the atom as a function.

If car of the form is an S-expression beginning with LAMBDA or NLAMBDA, the S-expression is the function. If CAR of the form begins with FUNARG, the funarg mechanism is invoked (see Section 11). Any other S-expression is evaluated.

If the evaluation produces NIL or the S-expression itself the form is faulty. Otherwise the result of the evaluation is treated as if it were car or the original form.

Finally, if car of the form is anything other than an atom or an S-expression the form is faulty and faulteval is called.



Note: variables c and d are for description only; they are not actually bound as variables.

Appendix 3

Control Characters

Several teletype control characters are available to the user for communicating directly to LISP, i.e., not through the read program. These characters are enabled by LISP as interrupt characters, so that LISP immediately 'sees' the characters, and takes the corresponding action as soon as possible. For example, control characters are available for aborting or interrupting a computation, changing the printlevel, returning to TENEX, etc. This section summarizes the action of these characters, and references the appropriate section of the manual where a more complete description may be obtained.

Control Characters Affecting the Flow of Computation

1.	control-H	(interrupt) at next function call, LISP goes into a break. See p. 16.3.
2.	control-B	(break) computation is stopped, stack backed up to last function call, and a break occurs. See p. 16.3.
3.	control-E	(error) computation is stopped, stack backed up to last errorset, and NIL returned as its value. See p. 16.4, 16.7, 16.14.
4.	control-D	(reset) computation is stopped, control returns to evalqt.
5.	control-C	(TENEX) computation is stopped, control returns to TENEX. Program can always be continued without any ill effect with TENEX CONTINUE command, p. 2.4.

If typed during a garbage collection the action of control-B, control-E, and control-D is postponed until the garbage collection is completed.

Typing control-E and control-D causes LISP to clear and save the input buffer. Its contents can usually be recovered via the $ (alt-mode) command, as described in section 22.

I/O Control Characters

1.	rubout	clears teletype input buffer. For example, rubout would be used if the user typed ahead while in a garbage collection and then changed his mind. p. 2.5.
2.	control-O	clears teletype output buffer, p. 2.5, 14.14.
3.	control-P	changes printlevel. p. 14.14.
4.	control-A, Q	line editing characters, see pp. 2.5, 14.6, 14.19.
5.	control-R	causes LISP to retype the input line, useful after several control-A's, e.g.,

```
user types: ←DEFINEQ((LAMDA\A\DDA\A control-R
LISP types:  DEFINEQ((LAMB
```

Miscellaneous

1. control-T

(time)
prints total execution time for program, as well as its status, e.g.,

```
←RECLAIM()

GC: 8
 RUNNING AT 15272  USED 0:00:04.4 IN 0:00:39
1933, 10109 FREE WORDS
10109
← IO WAIT AT 11623  USED 0:00:05.1 IN 0:00:49
```

2. control-S

(storage)
changes minfs. See p. 10.15.

3. control-U

if typed in the middle of an expression that is being typed to evalqt, breakl or the editor, will cause the editor to be called on the expression when it is finished being read. See section 22.

INDEX

Names of functions are in upper case, followed by their arguments enclosed in square brackets [], e.g. ASSOC[X;Y]. The FNTYP for SUBRs is printed in full; for other functions, NL indicates an NLAMBDA function, and * a nospread function, e.g. LISTFILES[FILES] NL* indicates that LISTFILES is an NLAMBDA nospread function.

Words in upper case not followed by square brackets are other LISP words (reserved variable names, commands, messages, etc.).

Words and phrases in lower case are not formal LISP words but are general topic references.

PAGE
NUMBER(S)

PAGE
NUMBER(S)

PAGE
NUMBER(S)

PAGE
NUMBER(S)

PAGE
NUMBER(S)

PAGE
NUMBER(S)

PAGE
NUMBER(S)

PAGE
NUMBER(S)

PAGE
NUMBER(S)

PAGE
NUMBER(S)

PAGE
NUMBER(S)

PAGE
NUMBER(S)

PAGE
NUMBER(S)

PAGE
NUMBER(S)