SH20-2076-0

Installed User Program

LISP/370
Program Description/Operations Manual

Program Number: 5796-PKL

This manual is intended as a guide to the facilities and capabilities of LISP/370. As such, it contains principally reference material describing the functions available in the system.

PROGRAM SERVICES

During a specified number of months immediately following initial availability of each licensed program, the customer may submit documentation to the designated IBM location below when he/she encounters a problem which his/her diagnosis indicates is caused by a defect in the licensed program. During this period only, IBM through the program sponsor(s), will, without additional charge, respond to an error in the current unaltered release of the licensed program by issuing known error correction information to the customer reporting the problem and/or issuing corrected or notice of availability of corrected code. However, IBM does not guarantee service results or represent or warrant that all errors will be corrected. Any onsite program services or assistance may be provided at a charge.

WARRANTY

THE LICENSED PROGRAM DESCRIBED IN THIS MANUAL IS DISTRIBUTED ON AN 'AS IS' BASIS WITHOUT WARRANTY OF ANY KIND EITHER EXPRESS OR IMPLIED.

Central Service Location: IBM Corporation
Detroit West Automotive Branch office
7700 Second Avenue
Detroit, Michigan 48202
Attention: Mr. A. E. Polcha

Central Service Available Until: April 27, 1980

First Edition (March 1978)

A form for readers' comments has been provided at the back of this publication. If this form has been removed, address comments to: The Central Service Location.

CONTENTS

INTRODUCTION

This manual is intended as a guide to the facilities and capabilities of LISP/370. As such, it contains principally reference material describing the functions available in that system. It also contains a certain amount of tutorial material intended to provide some motivation or explanation for why certain operations are performed in the way they are.

This manual is not intended as a basic primer for LISP. For that purpose, the reader should consult another publication such as *Let's Talk LISP* by Laurent Siklossy (Prentice-Hall, 1976), *The Programmer's Introduction to LISP* by W. D. Maurer (Elsevier, 1972), or *LISP 1.5 Primer* by Clark Weissman (Dickenson 1967), all of which are textbooks presenting an introduction to LISP for the beginning LISP programmer. Other books, such as *Artificial Intelligence* by Patrick Winston (Addison-Wesley, 1977) and *Computational Semantics* by E. Charniak and Y. Wilks (Elsevier, 1976) contain chapters introducing LISP in the course of examining some of the application areas where LISP programs have been significant.

This LISP system was originally developed in the VM/CMS programming environment, and this has affected the structure and facilities included in the implementation. Nevertheless, we have tried to avoid any real dependencies on features unique to that environment. The MVS/TSO implementation has packaged all of the system interface routines into a separately assembled module, and extensions in this module to offer more sophisticated interface services should be relatively straightforward.

The version of LISP/370 documented here is one of a series of systems produced during the continuing development of LISP at the IBM Thomas J. Watson Research Center. This version was selected for submission as an Installed User Program because it has been used for more than a year at that site, during which time we feel that most major implementation errors have been detected and corrected. Since the time when this system was devised, our thoughts about several aspects of LISP have evolved, but implementation of these ideas is still in an experimental stage.

Use of a screen display console, such as the IBM 3270 series of devices, is recommended for program development in the LISP/370 system. Normal test and debugging activity profits greatly from the rapid display capability of these devices. Once a LISP application has been developed, the value of this type of terminal will depend upon the application itself rather than any characteristic of LISP/370.

HOW TO ACCESS LISP/370 FROM CMS

The programs and procedures for running LISP/370 are available on the 194 minidisk belonging to the VM userid LISP370. (It is possible that a disk other than 194 be used for LISP, but this description is written assuming a specific address in order to provide examples of the actual commands the user should enter.) The user must link to this disk before running LISP, using the following commands or some EXEC procedure (such as GIME or LINK-WAIT) which performs similar functions:

```
CP LINK LISP370 194 2B4 RR
ACCESS 2B4 N
```

Persons who frequently use LISP may wish to have their user directory extended to include this link, or may update their PROFILE EXEC files to perform the necessary link and access.

There is nothing magical about the virtual device address 2B4 used in the example above. Any convenient address may be used.

Once the necessary disk is accessed, LISP/370 may be loaded by invoking the EXEC procedure LISP370. This procedure loads the current LISP system, and leaves the user talking to a LISP EVAL-type supervisor. To leave LISP, type (FIN) to this top-level supervisor. Control returns to the LISP370 EXEC procedure, which then releases the storage used by the LISP system. Alternatively, execution of the LISP function (RET) will produce an immediate return to the caller of LISP. (RET) may be evaluated at any level, it need not be the top level supervisor, to leave LISP.

If the user has saved a LISP system file image by using the FILELISP function, he may load that saved system by specifying the file identifier as an argument of the LISP370 EXEC procedure. For example, if the LISP system file image is named ASK FILEIM, it may be invoked by the command:

```
LISP370 ASK FILEIM
```

When such a saved LISP system is invoked, execution will continue with a return from the function FILELISP which saved the system. This may or may not be the top-level supervisor which receives control when the default system LISP370 FILEIM is invoked, depending upon the manner in which that particular LISP system was saved.

There are several options available when loading a LISP/370 system. These options control the allocation of storage for the system being loaded.

The principal option allows specifying how much storage will be used by the LISP system (what is not used by LISP remains available to CMS for executing other programs, such as a context editor). This may be specified either as an actual amount of space, for example 1600K or 3M, or as a fraction of the CMS storage currently available, such as 85%. Thus, to allocate 90% of available storage to LISP, one may use this command:

LISP370 (90%

If an explicit specification of the amount of CMS storage to be used by LISP is made, it must be the first option specified. Other storage allocation options come in pairs, the first indicating which option follows, and the second the actual specification of amount of storage or percent of available storage, as indicated above.

When LISP is loaded, a large chunk of CMS free storage is reserved by means of the DMSFREE macro. This allocates space at the high end of the user area, immediately below the space already used by CMS for disk directories, et cetera. In an effort to avoid storage fragmentation, which may occur if additional storage requested by DMSFREE macros is allocated below the LISP region (leaving a large hole when the LISP storage is released), the LISP loader will release part of the storage it obtains from the high end of the LISP region. The hope is that this storage will be sufficient for whatever processes may require it during and after LISP execution. The amount of storage to be returned to CMS is controlled by the CMSHIGH= option, and may be specified as an absolute amount, or as a percentage of the free storage available when LISPGET is invoked.

The options NIL=, BPI=, and STACK= allow the user to control the allocation of free space to the various parts of LISP. BPI= refers to the space reserved for new binary program images linked into the LISP system as a result of a COMP370 function or loaded from a LISPLIB where they had been written by an earlier compilation. NIL= refers to that part of the LISP system containing communication cells for quoted objects and shallow binding cells. A shallow binding cell is required for all free variables referenced by a compiled program. For this purpose, other functions referenced from within a compiled program count as free variables.

This allocation is performed every time a LISP system is loaded. Thus, if it is found while running LISP that a larger stack space in desirable, it is possible to use FILELISP to preserve the current state of the LISP system, then to reload the saved system using the same, or a different allocation of storage. Note that, since the division of space is performed each time LISP is loaded, reloading a file image may change the amount of space allocated to the various purposes because the amount of space which is actively used has changed or the amount of space available from CMS has changed.

Note also that a percentage specification of CMSHIGH= refers to available CMS storage, while percentage specifications of other options discussed in the previous paragraph refer to available LISP storage, the difference between the amount of storage obtained from CMS and that required for all LISP's active data.

If LISP storage allocation options are given as absolute amounts, they are satisfied if possible, then percentage allocations are performed based upon the remaining available storage. A request for too much storage will yield an error message and LISP will not run.

As an example of the use of these options, the following command represents the default values assumed by LISPGET if none of these options are specified.

LISPGET (80% STACK= 100K BPI= 5% NIL= 4% CMSHIGH= 2%

At least one blank must follow the equal signs. If an invalid LISPGET syntax is used, a short message showing the correct command format is written on the user's console.

Some more information about LISPGET, STRTLISP and LISPFREE

The LISP370 EXEC makes use of two utility programs in addition to LISPGET. LISPGET is the first to be invoked, and it allocates a large piece of CMS storage into which it reads the first, small part of a LISP file image. The starting address and length of the storage area is recorded in the CMS nucleus, so that it my subsequently be retrieved by the programs STRTLISP and LISPFREE. LISPGET then exits, returning to its caller.

STRTLISP is invoked when the LISP system whose loading and initialization has been started by LISPGET is to actually receive control and start (or, more precisely, resume) executing. This division of function may be useful in cases where the user desires to allocate the storage to be used by LISP before invoking other programs which also demand storage from CMS, and might take a variable amount of storage large enough to preclude running LISP. A context editor is typically such a program.

When STRTLIST is invoked, if retrieves the location of the LISP region from the CMS nucleus, and branches to the entry point for LISP.

LISP may return to its caller either through (FIN), which implements a permanent exit from the LISP system, or through (RET), which leaves LISP in such a manner that a subsequent STRTLISP will resume LISP execution. It is possible, for example, to use LISPGET to load LISP, then enter a context editor, then invoke an EXEC procedure from the editor (if the editor supports this) which uses STRTLISP to resume execution in LISP. It is even possible to make LISP communicate with the editor, either through the CMS console stack, or some more efficient, direct mechanism, if the context editor supports it. The only precaution which must be observed in doing this sort of thing is to avoid stepping on one's tail. That is, if (OBEY ...) is used to leave LISP and enter a context editor, there must be a normal return from the context editor to LISP, not an attempt to invoke LISP again through STRTLISP.

LISPFREE does what its name implies, it returns the storage currently occupied by LISP to CMS. The cells in the CMS nucleus which contained the starting address and length of the LISP area are reset. To reinvoke LISP, it is then necessary to start with LISPGET again.

HOW TO ACCESS LISP/370 FROM TSO

See APPENDIX 1.

LISP/370 DATA TYPES

The following is intended to be an intuitive introduction to the various data objects supported by LISP/370. Formal rigor is surrendered in favor of an effort to impart a sufficient operational understanding of these LISP data objects to make the following sections describing the standard LISP functions easier to use. For the programmer, the information presented here should indicate the range of data types available in the LISP/370 system and allow him to make some reasonable selections for use in describing his problems.

It is common, when speaking of LISP data objects, to talk about a vector, or an identifier, or perhaps a list cell, when in fact the object being discussed is actually a *pointer* to that vector, identifier, et cetera. This practice is ubiquitous in the LISP community, and will be employed in this manual. Only in cases where it is vitally important to make a distinction will the more cumbersome form "pointer to a vector" be used.

The pointers used by LISP/370 are full words (32 bits) and are rich pointers. This means that in addition to a storage address, they contain (in their high-order byte) a code indicating the type of object they point to. The reason for having these rich pointers, which do consume more storage space than would otherwise be necessary, has to do with efficiency. Many of the frequently occurring LISP operations require arguments of a specified type. Since the result of an operation performed on an invalid type of argument may actually destroy the LISP system, checking the types of arguments is essential, and this checking may be more efficiently performed if the type code is part of the pointer.

While it doesn't occur very frequently, garbage collection is a very expensive operation because of the quantity of data it processes. Having type codes associated with pointers makes garbage collection more efficient.

To facilitate the process of garbage collection, pointer type codes are classified into two groups -- pointers to stored objects and pointers to non-stored objects. A type code having a high-order one bit indicates a stored object; a high-order zero bit indicates a non-stored object.

This dichotomy is an artifact of the garbage collector and is somewhat misleading for the programmer, as it classifies binary programs as non-stored objects.

Nevertheless, there is a distinction to be made between pointers which contain the address of stored data, and pointers which might be thought of as containing immediate data. In the latter case, the type code in the pointer indicates the value of this data object is stored in the pointer itself, not in some other storage location. For example, small integral numbers are stored as part of a pointer with an appropriate type code, while floating point numbers are always stored in a memory location whose address is part of a pointer with appropriate type code.

The significance of this distinction between immediate data and stored data affects the concepts of sharing and updating. Stored data may be updated, and if it is shared by several structures, the updated data will also be shared (that is, all of the sharing structures are

simultaneously updated). Immediate data is intrinsically non-sharable; therefore, in this sense it is not updatable.

PAIRS

A *pair* is a stored data object having two component objects which are referred to as the CAR component and the CDR component (for historical and compatibility reasons). The storage allocation for a pair is two contiguous full-words. Both of these words contain pointers. The CAR component occupies the first word; the CDR component occupies the second word. Since a pointer is used to represent any LISP data object, a pair is an association of two completely arbitrary LISP data objects.

Two basic functions are provided for selecting part of a pair. CAR and CDR applied to a pair return as their value the corresponding component of the pair.

The print representation of a pair is normally a left parenthesis followed by the print representation of the first element of the pair, a blank, a period, a blank, the print representation of the second element of the pair, and finally a right parenthesis. In certain cases, however, a simpler or more complex print representation is used. These abrogations of the above rule occur because of the desire for a more readable print representation for lists, or to explicitly show shared substructure.

LISTS

Lists are composite objects created from pairs by applying a conventional interpretation to the pair data type. Thus each pair is a list. The CAR component of the pair is interpreted as the first element of that list, and the CDR component of the pair is interpreted as the remainder of that list.

(Note: It is likewise possible to give an interpretation of pairs as trees or rooted directed graphs.)

The distinguished object NIL is used to denote an empty list. Thus, if the CDR of a pair is NIL, there are no remaining elements in that list.

Having NIL as its CDR component is only one way in which a pair may be the end of a list. If the CDR of a pair is any LISP data object other than a pair, that pair terminates a list.

For the purposes of functions which operate on lists, the CDR component of the pair terminating the list is not considered to be part of the list.

The print representation of a list is a modification of the representation of its component pairs as described above. This modification is intended to improve readability by eliminating some of the parentheses and divulging the sharing of data; however, the inclusion of some (or all) of the deleted parentheses is always acceptable in input data. When a pair is pointed to from the CDR of another pair in a list which is being printed, the separating period and blank of the

original pair and its terminating right parenthesis, and the initial left parenthesis of the pair pointed to, are not printed. In addition, when the terminating pair of a list has NIL as its CDR component, that NIL and the space, period and space which would separate it from the CAR value are not printed. This seems more complicated when described in words than when illustrated by example.

Thus, the list

```
(A . (B . (C . NIL)))
```

would appear as

```
(A B C)
```

when printed.

Since a pair is a perfectly reasonable element of a list, it is possible to create lists which include themselves, or parts of themselves, as elements. LISP/370 uses a general scheme for input/output which indicates the sharing of data. This sharing scheme, as well as other aspects of the LISP/370 input/output system, makes use of a *break character* which is defined in the standard system as percent (%). An input expression written:

```
%L1=(A . %L1)
```

generates a pair whose CAR component is a pointer to the identifier A and whose CDR component is a pointer to the pair itself. The list interpretation of this pair would be a circular list --effectively an infinite list of A's.

This sharing notation need not generate a circular list. For example, the expression:

```
(%L1=(A) %L1)
```

generates a list containing two elements. The first element is the list containing a single element -- the identifier A -- and the second element is another identical pointer. This is to be distinguished from the expression:

```
((A) (A))
```

which also generates a list of two elements, each of which is a list containing the single identifier A. In this case, however, the two elements are different pointers, although they point to equal (but separately stored) lists.

For purposes of accessing the elements of the list, both expressions are equivalent (but note that the list having the shared data requires less storage). These two lists are not equivalent with respect to updating. That is, the product of updating one may not be the same as the product achieved by the same updating operation applied to the other.

In general, if it is true of two structure that corresponding accesses yield equivalent values then it can be said that the structures are equivalent trees (see EQUAL function). If it is true that the products of some updating operation applied to two structures would leave them EQUAL, then the structures can be said to be equivalent rooted directed graphs (see UEQUAL function).

NUMBERS

LISP/370 operates on three basic types of numbers, and on several other types of numbers formed through composition of these basic types. A basic numeric data item may be an integer or a real (also called a floating point number, or simply a float). Integers are stored in one of two possible formats, depending upon their value. In the range -2^{26} to $2^{26}-1$ (–67,108,864 to 67,108,863), the *small integer* format is used (see Figure 1). This format stores the numeric value as part of a pointer address field, and so achieves greater efficiency in computation and storage than the *large integer* format (see Figure 2) which is used for all other integer values. All integers are stored exactly by LISP. The only limitation on size is the available space in the heap.

Real numbers are stored using System/370 double precision floating point format, yielding 53 to 56 bits of precision for the mantissa and a range of up to (about) 10^{74}. Real numbers are stored in a separate section of the heap used only for these data. This area is allocated at the high address end of the space reserved for the heap, and extends toward lower addresses as new real numbers are generated.

The print representation for a real number always includes a decimal point to distinguish reals from integer values. This decimal point must be preceded by at least one decimal digit, to avoid possible confusion with the period used in printing pairs. A minus sign may precede the first digit to indicate a negative value.

Both integer and real numbers may be followed by a decimal exponent formed by the letter E, a plus or minus sign (plus is optional), and the exponent magnitude expressed in decimal digits.

There are two parameters which control the way in which real numbers are translated into their print representations for output. **FUZZ** refers to a value used to define the intended precision of real number operations. Two real numbers, X and Y, are equal in the LISP system if

$$|\,|X| - |Y|\,| <= FUZZ * \text{maximum}\,(|X|, |Y|)$$

Insofar as printing a real number, X, is concerned, a character representation is generated for the value in the range

$$X-FUZZ*|X| \text{ to } X+FUZZ*|X|$$

which results in the shortest character string. This print representation may include an exponent, in which case there will be exactly one decimal digit before the decimal point, or in cases where the number of digits (exclusive of decimal point and a possible minus sign) needed

Small Integer Pointer Format:

0011	S G - -	- -

S is a sign bit (1 for negative value, in two's complement form); G is a guard bit (normally the same as the sign bit) used by the arithmetic routines for detecting overflow (indicating conversion to a large integer is required);
- represents a data bit which is part of the actual numeric value.

Note that a small integer is actually a (non-stored) pointer value. It is not a reference to another data object.

Figure 1

Large Integer Format:

LCBVTP	Vector Length in Bytes
0	Low-order Digit (radix 2^{31})
	.
	.
	.
0	High-order digit (radix 2^{31})

The structure pictured above defines the magnitude of a large integer. There are two pointer type codes which designate large integers; one indicates a positive large integer, the other indicates a negative large integer. Because these type codes are not in the class of vectors, it is not possible to select an element (digit) of a large integer with vector functions such as ELT.

Figure 2

to represent the numeric value is less than **NDIGITS**, no exponent will be printed and the decimal point will be placed wherever is required.

The user may specify values for FUZZ and NDIGITS by using the function SETFUZZ.

Reference Vector Format:

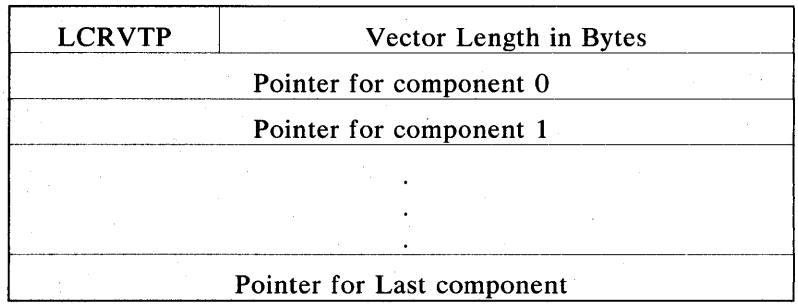


Figure 3

VECTORS

LISP/370 vectors may be classified into two general types: pointer vectors and non-pointer vectors. Pointer vectors, as the name implies, may contain references to any LISP data objects (including themselves, so circular structures are possible). Pointer vectors are further classified as *reference vectors* and *selector structures.*

Non-pointer vectors contain binary information -- that is, data which cannot contain references to other data objects. Thus, non-pointer vectors are non-descendable from the point of view of the garbage collector and structure-dependent functions such as EQUAL and PRINT. Non-pointer vectors are further classified as *bit vectors*, *character vectors*, *word vectors* and *float vectors*.

Except for bit vectors, vectors may have any length for which sufficient space exists in the heap. Bit vectors may have a maximum of 2^{24}-1 (16,777,215) elements (bits).

All vectors use zero-origin indexing for identification of their components. The function ELT is a general vector accessing function, applicable to any type of vector. Thus

(ELT vector 0)

is always the first element of vector. Other accessing functions, tailored to a particular type of vector, are provided because they are more efficient in execution, or because a more specific check on the type of argument is desired. These are described in the section on vector functions.

The print format of a reference vector uses angle brackets to delimit the extent of the vector and blanks to separate elements of the vector:

$<comp_0\ comp_1\ \ldots\ comp_n>$

Selector Structure Format:

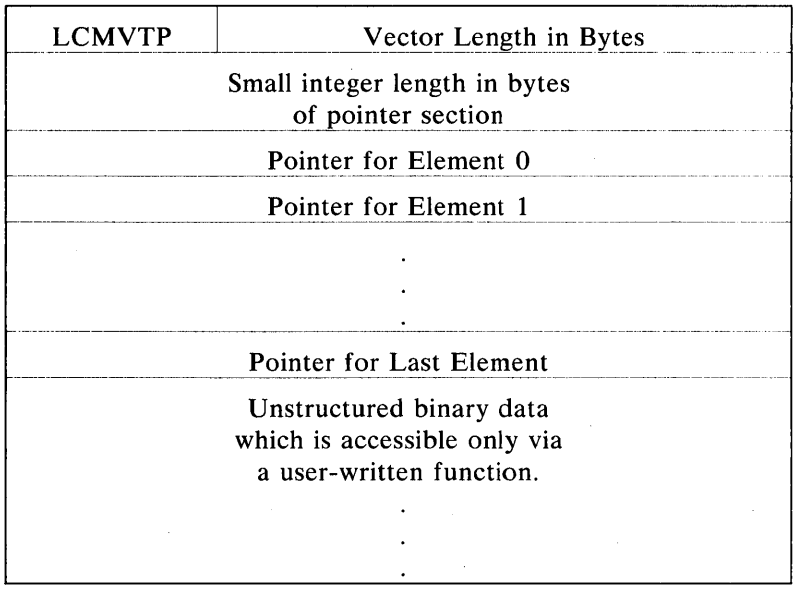


Figure 4

where ***comp***$_n$ is the print representation of the LISP data object referenced as the n'th element of the reference vector.

Selector structures present a more difficult problem for printing, because there is no standard organization of the binary data section of a selector structure. Therefore, the print representation of a selector structure is:

%S<***comp***$_0$ ... ***comp***$_n$ %X'***hex***...***hex***'>

where ***comp***$_n$ again indicates the print representation of an object referenced in the pointer section of a selector structure. '***hex***...***hex***' is the binary data part of the selector structure, printed as hexadecimal ***hex*** characters.

Character Vectors

Strings (character and bit vectors) share a special storage characteristic in the LISP/370 system. For reasons of economy (of both storage and processing time) they are stored in contiguous blocks of storage. Nevertheless, because it is considered desirable to allow them to vary in length, a compromise has been achieved which involves maintaining two separate

Character Vector Format:

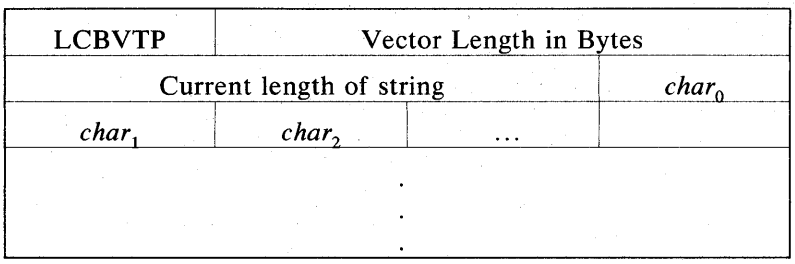


Figure 5

pieces of length information for each string. One length reflects the amount of storage allocated for the string, in terms of the number of elements which may be put into the string without having to allocate more storage for a larger string. The other length refers to the current number of elements which are actually used, which is less than or equal to the capacity of the string.

There are two input/output representations for character vectors. The more general format is:

%**k'*char*...*char*'**

where **'k'** is the maximum number of characters which could be put into the vector for the character string being read or printed (see the figure depicting string formats below). The actual contents of the character string **'*char*...*char*'** reflects only the current length of the string, and might be null. Any character may be included as part of a character string; however, the string delimiter character and the letterizer character must be treated specially. In order to avoid confusion about whether a string delimiter character actually delimits a string or is intended as a data character in a string, every occurrence of the string delimiter character as a data character in a string must be prefixed by a letterizer character. This letterizer character is not part of the character string in storage; it is created during output by the print routine, and discarded during input by the read routine. Likewise, every occurrence of the letterizer character as a data character in a character string must be prefixed by the letterizer character.

For example, the string

'|''

contains one character (a string delimiter), and the string

'|||''

contains two characters (a letterizer and a string delimiter).

When it is necessary to represent a character string whose total capacity is not larger than the shortest vector necessary to contain the characters specified, the simpler form:

'char...char'

may be used. This designates a character vector which may have zero, one, two or three unused elements. Referring to Figure 5, it may be seen that if N is the number of real characters in a string (letterizing characters are not counted), the number of unused elements for this simplified notation is residue (N-1):4.

Example: to specify an eight-element character vector containing the letters 'FUNCTION', write:

'FUNCTION'

This vector will have space for nine characters (see Figure 5) and a current length of eight. To specify a vector with a capacity of 100 characters, but with a current length of zero, write:

%100''

Bit Vectors

The input/output format of bit vectors is similar to the format for character vectors; however, 4-bit segments are represented by one hexadecimal character and the current length field is a count of the number of bits in the vector, not a count of the number of bytes (see Figure 6). Only the characters 0...9 and A...F may be specified as part of a bit string.

There are variant input/output representations for bit vectors, depending upon the current length of the vector being considered. For bit vectors whose length is a multiple of four bits, the format is:

%B**k**'***hex...hex***'

where **'k'** is the maximum number of bits which the specified vector could contain. The actual contents of the bit string ***'hex...hex'*** reflects only the current length of the string, and might be null.

Bit Vector Format:

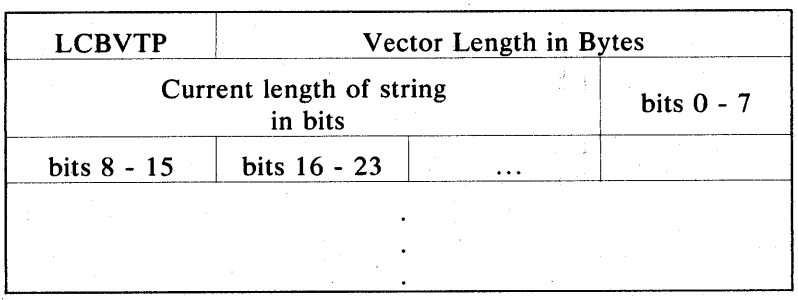


Figure 6

As with character vectors, the maximum length field is optional and may be omitted when representing a vector of length consistent with the explicitly specified data. A bit vector specified without an explicit maximum length **'k'** and with up to 28 unused elements has the format:

%B'***hex...hex***'

For bit vectors whose current length is not a multiple of four bits, the format is:

%B**k**:**c**'***hex...hex***'

where **'k'** is as previously defined and **c** is the current number of bits in the string. A bit vector specified without a maximum **'k'**, but with a current length **'c'** and with up to 31 unused elements has the format:

%B:**c**'***hex...hex***'

FUNARGS

A FUNARG is a expression closure -- that is, the combination of a expression definition with a specific environment in which that expression is to be executed. It is represented as a list of three elements:

(FUNARG ***expression sd***)

where the first element is the identifier FUNARG, the second element is the actual expression, which will evaluate to an object which may be applied) and the third element is a state descriptor which defines the environment.

WARNING: while it is possible to manipulate FUNARGS as if they were lists, the user is strongly advised to refrain from this practice. There are two reasons for this:

(1) Binary programs are compiled with an understanding of their immediate environment. By executing a BPI in another environment, unpredictable action (including failure of the LISP/370 system) may occur.

(2) In view of (1), it is planned to make FUNARGS a special data type in the future, at which time list processing functions will not accept them as arguments.

STATE DESCRIPTOR

A state descriptor is an elementary data object generated by the STATE basic macro. It is conceptually a pointer to a particular stack frame, which serves to define either an environment (a set of identifier -value associations) or a previous state, which denotes a specific point in the application of a FUNARG. Practically, state descriptors have the capacity to contain some control information, since this is required by the garbage collector and by their use to determine validity of shallow bindings. Thus they are five-word objects and are processed only by a limited set of functions which are prepared to maintain their structure.

Creation of a state descriptor ensures that the related stack frame will be retained until the state descriptor is deleted by the garbage collector when there are no references to it.

State descriptors serve two purposes. First, they define an environment which may be used to create function closures. Second, they are actually saved states which may applied in order to effect a transfer from the current state to the saved state. Execution will subsequently proceed in the environment of the saved state, at the point immediately following the STATE operation which created the saved state. When a state descriptor is applied, it must have an argument, which is evaluated in the environment initiating the application. The value resulting from this evaluation becomes the value of STATE when execution resumes in the saved state.

BINARY PROGRAM IMAGES

It is not possible to print binary program images in a form which would permit them to be subsequently read by LISP and used like the original object. There are several reasons for this, the major difficulty being the relationship between the binary program and the entire LISP system, which makes the same program printed at one time from a particular LISP system incompatible with another LISP system, or possibly even with the same LISP system at a different point in time.

Therefore, since it frequently occurs that an object being printed contains references to binary programs (e.g. in a backtrace), a convention is used which incorporates the name of a binary program (that is, the identifier associated with the BPI when it was compiled) in the form:

%SUBR.BPINAME or %MSUBR.BPINAME

where SUBR is used for functions with evaluated arguments, and MSUBR is used for macros (functions with unevaluated arguments).

If an attempt is made to read such a form, the read program will emit an error message and use the .NOVAL object instead of a binary program.

STANDARD FUNCTIONS

The discussion of standard functions is organized into several sections according to the intended application of the functions being described. Standard, in this context, means the function which is supplied in the LISP/370 system as supported by the Yorktown Computing Center. Most of these functions may be redefined by the user should his requirements demand it. Some applications must do this (generally those concerned with programs written for a foreign LISP system), but it is hoped that the user will find the standard versions acceptable. Their use will contribute to the exchange of programs between applications and generally aid in the understanding and debugging of programs throughout the LISP community.

In each of the following sections describing various groups of functions, the functions are treated alphabetically.

BASIC FUNCTIONS AND MACROS

Basic function and *basic macro* are terms used to indicate operations whose meanings have been built into the LISP/370 system in such a manner as to make them immutable. There are two reasons for having any such functions at all. The first is that there comes a time when an operation cannot be readily defined in terms of more primitive operations, but instead is said to have an understood meaning. Basic macros are commonly of this sort. The second reason is a consideration of efficiency. Particularly in the compiler, certain optimizations can be made based upon the knowledge that an operation's meaning and implementation is unchanging.

Basic functions are described in the sections which follow, along with other functions having similar applications. All of the basic functions and macros are listed below for ease of reference.

The identifiers designating basic functions and basic macros have different type codes than other identifiers. The function HEXEXP prints the hexadecimal representation of its (pointer) argument value, and may be used to exhibit the difference to a conversational user. Inside a function, the type testing functions FRP and MRP will test if their argument is a basic function or basic macro, respectively.

Basic Functions

APPLX	FLOATP	NUMBERP
ATOM	FRP	PAIRP
BITSTRINGP	GENSYMP	PLEXP
CAR	IDENTP	RPLACA
CDR	LINTP	RPLACD
CONS	LISTP	SET
EQ	MDEFX	SMINTP
EVA1	MRP	STATEP
EVAL	NTUPLEP	STRINGP
FIXP	NULL	VECP

Basic Macros

*CODE (obsolescent: function is performed by application of FR*CODE expression)

COND	LAMBDA
EXIT	MLAMBDA
FR*CODE	QUOTE
FUNARG	RETURN
FUNCTION	SEQ
GO	SETQ
LABEL	STATE

STANDARD LIST FUNCTIONS

(APPEND list item)

If the argument **list** is non-pair, the value of APPEND is **item**. If **list** is a list, then a top-level copy of **list** is made, and the final CDR of the copied list is set to **item**. The value of APPEND then becomes this copied list. If **list** is circular, the function will loop.

A top-level copy means that new pairs are CONSed into a list, with their CAR components being the corresponding values from the list **list**. There is no copying of the structure below this top level, which would be accessed by descending the elements of **list**.

(ATOM x)

This function returns the value NIL if **x** is a pair; otherwise, value is *T*.

It is unfortunate that the word ATOM is wasted as the not-pair predicate, but to change this tradition would lead to considerable confusion and problems of compatibility.

(CAR x)

One of the two basic selection functions defined on pairs. Its value is the CAR component of the pair **x**. In its list interpretation, the value of (CAR **x**) is the first element of the list **x**.

If **x** is not a pair, an error results.

(C...R x)

There are sixteen macros defined in LISP/370 which give meaning to expressions of this form, where ... designates any sequence of one to four As or Ds. For example, (CADDR **x**) is equivalent to

(CAR (CDR (CDR **x**)))

and so on. This is literally true for interpreting such a macro, but if the macro is expanded by the LISP compiler, it is smarter than that and achieves the complete operation with only one call to an appropriate subroutine. In fact, this subroutine is capable of handling strings of CARs and CDRs up to 256 levels deep, and the macro will economize (CADR (CDR **x**)) to (CADDR **x**), and will even do well by (C..R (C...R **x**)) where .. is two, three or four letters, and ... is any number such that total

depth is less than 256.

(CDR **x**)

One of the two basic selection functions defined on pairs. Its value is the CDR component of the pair **x**. If **x** is not a pair, an error results.

The value of (CDR **x**) is usually the list containing all but the first element of the list **x**; however, this is not the case when **x** is the terminating pair of a list. In that case, (CDR **x**) is the terminating atom, usually NIL.

The use of the words CAR and CDR is also one of those laments of history, and while FIRST and REST might be preferred (and may be defined by the user), a certain tenacious tradition causes the use of CAR and CDR to thrive.

(CONC $\mathbf{list}_1$ ... $\mathbf{list}_n$)

This is a macro which expands into an expression which uses APPEND to create a list from several lists. It will accept an arbitrary number of lists as arguments. For example,

(CONC $\mathbf{list}_1$ $\mathbf{list}_2$ $\mathbf{list}_3$)
= (APPEND $\mathbf{list}_1$ (APPEND $\mathbf{list}_2$ $\mathbf{list}_3$))

Note that the last argument to CONC appears as the second argument to APPEND, so that it is not copied at the first level as are the other arguments to CONC, which appear as first arguments to APPEND.

(CONS **x y**)

CONS is the basic list-forming function. Its value is the new pair constructed with **x** as its CAR component and **y** as its CDR component. Of particular interest is the case where **y** is a pair. Then the value of CONS is the list formed by adding **x** to the beginning of the list **y**.

(EFFACE **item list**)

Uses EQUAL to search **list** for the first occurrence of **item**. If **item** is not found, or if **list** is atomic, returns **list** as its value. If **item** is found, it is removed from **list** by updating via RPLACD, and the updated list is returned as the value of EFFACE. Only the first occurrence of **item** will be removed from **list**.

(EQSUBSTLIST **newlist oldlist structure**)

This function substitutes corresponding elements of **newlist** for those elements of **oldlist** occurring in **structure**. The original **structure** is unchanged and EQSUBSTLIST constructs and returns a new object having the same form as **structure**. **structure** is searched recursively, atom by atom, using the EQ test, for each element of **oldlist**; therefore, only atoms are meaningful as elements of **oldlist**. It is possible to replace an atom in **structure** with any expression, including a list, in **newlist**; however, the replacing value from **newlist** is not scanned for other possible replacement. If the same (that is, EQ) atom appears more than once in **oldlist**, only the first occurrence is meaningful.

If **newlist** and **oldlist** are both NIL, EQSUBSTLIST returns a copy of **structure** as its value. Otherwise, **newlist** and **oldlist** must both be pairs and must be of the same length. If **structure** is circular, the function will loop.

The value of EQSUBSTLIST is a new list containing old atoms -- that is, new pairs are allocated to contain pointers to the existing objects in **structure** or **newlist**. Note the exception indicated above, however, if any elements of **newlist** are pairs.

(INTERSECTION $\mathbf{list_1}$ $\mathbf{list_2}$)

Constructs a new list containing only those elements appearing (at the top level) in both $\mathbf{list_1}$ and $\mathbf{list_2}$. The order of elements in this new list is the reverse order of their occurrence in $\mathbf{list_1}$. If either argument is not a pair, it is treated as if it were the only element in a list of length one.

(LAST **list**)

Returns as value the last element of **list** (i.e. the CAR of the last pair comprising **list**). If **list** is non-pair, the value of LAST is 0.

This function will loop if given a circular list as an argument. (See also LASTNODE.)

(LASTNODE **list**)

This function returns as its value the last pair forming the list **list**. It will loop endlessly if **list** is circular. If **list** is non-pair, an error break is taken. (See also LAST.)

(LENGTH **x**)

If **x** is a pair, LENGTH returns the number of elements in the list beginning with that pair. If **x** is not a pair, the value of LENGTH is zero.

NOTE : SIZE is the general function which computes the length of vectors as well as lists.

(LIST e_1 ... e_n)

Returns as value a list of n elements, the first element being the value of the expression e_1, et cetera.

(LISTOFSAME **list**)

Returns NIL if **list** isn't a pair, or if the final CDR of **list** is not NIL, or if the elements of **list** have differing type codes. Otherwise, returns true.

(LISTP **x**)

This function returns the value *T* is **x** is a list; otherwise, it returns the value NIL. The value of LISTP applied to NIL (the empty list) is *T*.

(MAP **list funct**)

This is the historical LISP MAP function which applies **funct** to **list**, then to (CDR **list**), then to (CDDR **list**), et cetera, until (CD...R **list**) is not a pair. The value of MAP is **list**, the original argument.

In LISP/370, MAP is implemented by a macro which invokes the more general operation MMAP by creating the expression (MMAP **funct list**). See the discussion of MMAP.

(MAPCAR **list funct**)

This is the historical LISP MAPCAR function, which applies **funct** to (CAR **list**), then to (CADR **list**), then to (CADDR **list**), et cetera, until (CD...R **list**) is non-pair. As the values of **funct** applied to the various elements of **list** are computed, they are CONSed into a new list which becomes the value of MAPCAR. The first element of this new list is the value of (**funct** (CAR **list**)), the next element is the value of (**funct** (CADR **list**)), et cetera.

In LISP/370, MAPCAR is implemented by a macro which generates the expression (MMAPCAR **funct list**) invoking the more general MMAPCAR operation. See also the discussion of MMAP.

(MAPLIST **list funct**)

This is the historical LISP MAPLIST function which applies **funct** to **list**, then to (CDR **list**), then (CDDR **list**) until (CD...R **list**) becomes non-pair. As **funct** is applied to the consecutive tails of **list**, the values issuing from those applications are CONSed into a new list which becomes the value of MAPLIST. The first element of this new list is the value of (**funct list**), et cetera.

In LISP/370, MAPLIST is implemented as a macro which invokes the more general MMAPLIST operation by creating the expression (MMAPLIST **funct list**). See also the discussion of MMAP.

(MEMBER **item list**)

This function searches the list **list** for the object **item**, using EQUAL testing for identity. It **item** is not found, or if **list** is not a pair, the value of MEMBER is NIL. If **item** is found, the value of MEMBER is that portion of **list** beginning with **item**. **list** is searched on the top level only.

(MEMQ **item list**)

This function is similar to MEMBER, except that it uses EQ testing for identity instead of EQUAL testing.

(MMAP **funct** $\mathbf{list}_1 \ldots \mathbf{list}_n$)

Establish an iteration which evaluates (**funct** $\mathbf{list}_1 \ldots \mathbf{list}_n$), (**funct** (CDR $\mathbf{list}_1$) ... (CDR $\mathbf{list}_n$)), ... (**funct** (CD...R $\mathbf{list}_1$) ... (CD...R $\mathbf{list}_n$)). The iteration stops whenever the CDR of any **list** becomes non-pair, which is to say that the number of iterations is equal to the number of items in the shortest list. If there are zero iterations (one of the **list** arguments was non-pair), the value of MMAP is NIL. Otherwise, the value of MMAP is the original argument $\mathbf{list}_1$.)

In LISP/370, the MMAP... functions are implemented by macros which generate in-line a PROG expression implementing the iteration over $\mathbf{list}_1$, ... $\mathbf{list}_n$. In the macro-generated expression, **funct** appears in operator position in order to effect the necessary function application. Because of this, it is possible to specify **funct** as an unquoted lambda expression yet avoid the construction of a FUNARG which would happen in the case of a lambda expression evaluated as the operand of a function. It also allows specifying a macro or MLAMBDA expression as **funct**, since the normal LISP/370 operator evaluation semantics will be used when looking at **funct** instead of operand evaluation semantics.

(MMAPC **funct** $\mathbf{list}_1$... $\mathbf{list}_n$)

This is similar to MMAP, except that (**funct** (CAR $\mathbf{list}_1$) ... (CAR $\mathbf{list}_n$)), ... (**funct** (CAD...R $\mathbf{list}_1$) ... (CAD...R $\mathbf{list}_n$)) is evaluated. MMAPC is implemented by a macro in LISP/370 (see discussion of MMAP).

(MMAPCAN **funct** $\mathbf{list}_1$ ... $\mathbf{list}_n$)

This is similar to MMAPCAR, except that the values computed by **funct** are NCONCed together, rather than CONSed into a list. For example,

(MMAPCAN **list** (QUOTE (1 2)) (QUOTE (3 4)))
= (1 3 2 4).

This function is implemented by a macro in LISP/370 (see the discussion of MMAP).

(MMAPCAR **funct** $\mathbf{list}_1$ ... $\mathbf{list}_n$)

An iteration is established which evaluates (**funct** (CAR $\mathbf{list}_1$) ... (CAR $\mathbf{list}_n$)), (**funct** (CADR $\mathbf{list}_1$) ..., (CADR $\mathbf{list}_n$)), et cetera, until (CD...R **list**) is non-pair, which is to say that the iteration continues for the number of elements in the shortest **list**. If any **list** is non-pair, the value of MMAPCAR is NIL. Otherwise, a new list is made as the iteration progresses, whose first element is (**funct** (CAR $\mathbf{list}_1$) ... (CAR $\mathbf{list}_n$)), et cetera. For example,

(MMAPCAR **list** (QUOTE (1 2)) (QUOTE (3 4)))
= ((1 3) (2 4)).

This function is implemented in LISP/370 by a macro (see the discussion of MMAP).

(MMAPCON **funct** $\mathbf{list}_1$ ... $\mathbf{list}_n$)

This function is similar to MMAPLIST, except that the values computed by **funct** are NCONCed together, rather than being CONSed into a list. For example,

(MMAPCON **list** (QUOTE (1 2)) (QUOTE (3 4)))
= ((1 2) (3 4) (2) (4)).

This function is implemented by a macro in LISP/370 (see the discussion of MMAP).

(MMAPLIST **funct** $\mathbf{list}_1$ ... $\mathbf{list}_n$)

This function is similar to MMAP, except that the value returned is a new list made by CONSing the values computed by **funct**. The first element of the value list is (**funct**

list$_1$ ... **list**$_n$), et cetera. For example,

(MMAPLIST **list** (QUOTE (1 2)) (QUOTE (3 4)))
= (((1 2) (3 4)) ((2) (4))).

(NCONC **list item**)

If **list** is non-pair, the value of NCONC is **item**. If **list** is a circular list, this function will loop indefinitely. Otherwise, RPLACD is used to replace the final CDR of **list** with **item**, and the updated **list** is returned as the value of NCONC.

(NREVERSE **list**)

This function shuffles the CDR components of the pairs making up **list** so that the CDR of the last pair in **list** points to the nex-to-last pair, et cetera. The CDR of the first pair is made NIL, and the value of NREVERSE is the last pair of **list**, which is now the first pair of a list containing exactly the same elements as **list**, but in reversed order. See also REVERSE, which constructs a new list in reversed order instead of reusing the pairs in the original list.

(NULL x)

This function has the value *T* if x is NIL; otherwise, it returns the value NIL.

(PAIRP x)

This function returns x if x is a pair; otherwise, it returns the value NIL. It is distinguished from LISTP by the fact that (LISTP NIL) = *T*, whereas (PAIRP NIL) = NIL.

(REVERSE **list**)

This function returns as its value a new, top-level copy of the list **list** where the elements of this new list are in the inverse order of their occurrence in **list**. See also NREVERSE.

(RPLACA x y)

This is one of the two basic functions for updating pairs. Its value is the updated pair which results when the CAR component of the pair x is replaced by y. An error is indicated if x is not a pair.

(RPLACD **x y**)

This is the other basic function which updates pairs. Its value is the updated pair which results when the CDR component of the pair **x** is replaced by **y**. An error is indicated if **x** is not a pair.

(SELECT e'_0 (e'_1 e_1) ... (e'_n e_n) e_0)

SELECT evaluates the e_n following the first e'_n EQ to the value of e'_0. It is a macro which generates the expression

```
((LAMBDA (%Gn) (COND ((EQ %Gn e'1) e1) ...
                     ((EQ %Gn e'n) en)
                     (1 e0)))  e'0)
```

where %Gn is a unique GENSYM and e_0 must appear. e_n may be a SEQ or PROGN sequence.

(SUBST **new old list**)

This function substitutes **new** for **old** in **list**. The original **list** is unchanged and SUBST constructs and returns a new object having the same form as **list** with each element EQUAL to **old** replaced by the value of the argument **new**.

If **new** and **old** are both NIL, SUBST returns a copy of **list** as its value. IF the value of **list** is not a pair, then the value of SUBST will be **list**. No substitution is done and no new structure is created. If there are no occurrences of **old** in **list**, the effect is to make a top-level copy of **list**.

See also EQSUBSTLIST.

(UNION $\mathbf{list}_1$ $\mathbf{list}_2$)

This function constructs a new list contains all the elements appearing in either of the lists $\mathbf{list}_1$ and $\mathbf{list}_2$. Each element appears only once in the value list. MEMBER is used to detect whether an element appears in a list.

STANDARD STRING FUNCTIONS

(ANDBIT **str_1** **str_2** ... **str_n**)

ANDs the bit strings **str_1** ... **str_n** and returns as value the resultant string. The operation is performed left to right: the result of the AND of **str_1** and **str_2** is ANDed with **str_3**, that result is ANDed with **str_4**, and so on. None of the argument strings is changed by the operation.

An error is indicated if the strings are not equal in length.

(BITGREATERP **str_1** **str_2**)

Compares two bit strings, and returns true if **str_1** is greater than **str_2**. If the strings are unequal in length, the shorter string is considered to be padded on the right with zeros for purposes of comparison. The argument strings are not changed by this function, even the bits beyond the current length of the strings are preserved.

If **str_1** or **str_2** is not a bit string, an error break occurs.

(BITSTRINGP **x**)

This function returns the value **x** if **x** is a bit string (i.e. a vector of bits); otherwise, it returns the value NIL. This is a basic function and is therefore not redefinable.

(CHANGELENGTH **str n**)

The length of the string **str** is updated to be the value **n**. An error is indicated if **str** is not a character or bit string, or if the value of **n** exceeds the maximum potential length of **str**. The value of CHANGELENGTH is the string **str** with its new length.

(FETCHCHAR **str n**)

Returns as value the character object (identifier whose print name is a single character) corresponding to the **n**th element of the character string **str**. An error is indicated if **str** is not a string, or if **n** is negative or exceeds the current length of the string **str**.

This function is slightly more efficient than the general vector selection function ELT when the argument is a character string.

(GETBITSTR x)

Allocates a bit vector with a capacity of at least **x** bits. The new vector is returned as the value of GETBITSTR. Vectors are allocated in increments of full words: for a bit vector, the first word includes only the first 8 bits of the string, prefaced by a 3-byte current length field (see Figure 6). Therefore, the actual capacity of the vector is defined by

(((x + (31 +(3*8))) / 32) * 32) – 24 bits.

x	*Maximum Capacity of Allocated Vector*
1-8	8
9-40	40
41-72	72
...	...

(GETFULLSTR **length fill**)

Similar to GETSTR in that a new character vector is allocated and returned as the value of GETFULLSTR. The new string, however, contains **length** instead of zero characters. The **fill** argument is optional. If it is specified as an identifier, the new string will be initialized so that each character is the initial letter of the P-name of **fill**. If **fill** is not specified, the string will be initialized to blank characters.

(GETSTR x)

Allocates a character vector with a capacity of at least **x** characters. The new vector is returned as the value of GETSTR. Vectors are allocated in increments of full-words: for a character vector, the first word includes only the first character of the string, prefaced by a 3-byte current length field (see Figure 5). Therefore, the actual capacity of the vector is defined by

(((x + 6) / 4) * 4) – 3 characters.

x	*Maximum Capacity of Allocated Vector*
1	1
2-5	5
6-9	9
...	...

Zero or negative numbers are invalid values for **x** and will cause an error break. The character string returned by GETSTR is initialized to the null string.

(IDENTP **x**)

This function returns the value **x** if **x** is an identifier; otherwise, it returns the value NIL. Note that there is a possibility for confusion in one case: (IDENTP NIL) = NIL, because NIL is actually an identifier.

This is a basic function, therefore it is not redefinable.

(ORBIT $\mathbf{str}_1$ $\mathbf{str}_2$ ... $\mathbf{str}_n$)

ORs the bit strings $\mathbf{str}_1$ ... $\mathbf{str}_n$ and returns as value the resultant string. The operation is performed left to right: the result of the OR of $\mathbf{str}_1$ and $\mathbf{str}_2$ is ORed with $\mathbf{str}_3$, that result is ORed with $\mathbf{str}_4$, and so on. None of the argument strings is changed by the operation.

An error is indicated if the strings are not equal in length.

(RPLACSTR $\mathbf{str}_1$ $\mathbf{index}_1$ $\mathbf{len}_1$ $\mathbf{str}_2$ $\mathbf{index}_2$ $\mathbf{len}_2$)

This is an efficient, generalized string modification routine. It can replace any part of $\mathbf{str}_1$ with any part of $\mathbf{str}_2$, making any necessary adjustment in the length of $\mathbf{str}_1$ because the replacement characters from $\mathbf{str}_2$ are greater or fewer than the characters being replaced in $\mathbf{str}_1$. Furthermore, it can insert $\mathbf{str}_2$, or some specified substring of $\mathbf{str}_2$, into $\mathbf{str}_1$.

$\mathbf{str}_1$ must be a character vector, else an error is indicated. $\mathbf{str}_2$ may be either a character vector, or it may be a stored identifier (not a GENSYM). In the latter case, the print name of the identifier (which is a character string) will be used as $\mathbf{str}_2$, and $\mathbf{index}_2$ and $\mathbf{len}_2$ refer to this string. If $\mathbf{str}_1$ or $\mathbf{str}_2$ are not ad described, an error is indicated.

$\mathbf{index}_1$ specifies the index of the first character in $\mathbf{str}_1$ to be replaced. $\mathbf{len}_1$ specifies the number of consecutive characters, beginning with the $\mathbf{index}_1$ character, to be replaced. $\mathbf{index}_2$ and $\mathbf{len}_2$ specify the location and number of characters from $\mathbf{str}_2$ which are to replace the designated characters in $\mathbf{str}_1$.

$\mathbf{index}_1$, $\mathbf{index}_2$, $\mathbf{len}_1$ and $\mathbf{len}_2$ may be either integer values or NIL; an error is indicated if they are not.

In general, an **index** may vary from zero to the current length-1. If NIL is specified for an index, the numeric value zero is used. If $\mathbf{index}_1$ is equal to the current length, $\mathbf{len}_1$ must be zero: by use of this convention, $\mathbf{str}_2$ can be appended to the end of $\mathbf{str}_1$.

If zero is specified for $\mathbf{len}_1$, $\mathbf{str}_2$ is inserted in $\mathbf{str}_1$ before the position specified by $\mathbf{index}_1$. If NIL is specified for a length, all of the characters from the related index value to the end of the string are used. In effect, using NIL for the value of $\mathbf{len}_x$ is an efficient way of specifying the value:

(DIFFERENCE (STRINGLENGTH **str**$_x$) **index**$_x$)

len$_2$ and **index**$_2$ are optional arguments. If they are not specified, a value of NIL will be assumed.

Whenever possible, RPLACSTR will update the original **str**$_1$, and return as its value the updated string. However, if **len**$_2$ is greater than **len**$_1$, it is possible that **str**$_1$ does not have sufficient space for the result string. In this case, a new string is constructed and this new string is returned as the value of RPLACSTR.

The user may test whether the updated string is the original **str**$_1$ or a copy by an expression such as:

(EQ **str**$_1$ (SETQ **temp** (RPLACSTR **str**$_1$ ...)))

which will be true if **str**$_1$ has been updated in place, and false if a new string had to be created. The purpose of the SETQ operation is to preserve the value of RPLACSTR in case a new string was created.

(STORECHAR **str n chr**)

Updates the character string **str** by replacing the **n**th character with the first character of the print name of the stored identifier **chr**. An error is indicated if **str** is not a character vector, or if **n** is negative or exceeds the current length of the string **str**, or if **chr** is not a regular identifier.

The value of STORECHAR is the last argument to STORECHAR, the value used to update the designated character of the string.

(STRCONC **str**$_1$ ... **str**$_n$)

This function returns as its value a new string made by concatenating all of the strings **str**$_1$ ... **str**$_n$. **str**$_x$ may be either a character vector or a stored identifier (not a GENSYM). In the second case, the print name of the identifier is concatenated into the result string. If **str**$_x$ is not a character string or stored identifier, an error is indicated.

(STRGREATERP **str**$_1$ **str**$_2$)

This functions compares two character strings and returns true if **str**$_1$ is greater than **str**$_2$, otherwise it returns NIL.

If the two strings are of unequal length, the shorter string is considered to be padded on the right with binary zeros for purposes of comparison. If an argument is not a character string, an error break occurs.

(STRINGLENGTH **str**)

This function returns as its value the current length of the character or bit string **str**. An error is indicated if **str** is not a character or bit string.

(STRINGP **x**)

This function returns the value **x** if **x** is a character string (that is, a vector of characters); otherwise, it returns the value NIL. This is a basic function, therefore it is not redefinable.

(SUBSTRING **string index length**)

This macro returns a copy of part (or all) of the character string STRING. The returned value starts with the **index** character of STRING (remember, index zero is the first character) and is **length** characters long. If **length** is specified as (), that designates the end of the string.

(SUFFIX **chr str**)

Updates the character string **str** by adding the first character of the print name of the stored identifier **chr** to the end of the string. **chr** is usually a character object, but may be any stored identifier. This function increments the length of the character string by one, or causes an error break if there is not sufficient space in the string **str** for this additional character.

(XORBIT $\mathbf{str}_1$ $\mathbf{str}_2$ ... $\mathbf{str}_n$)

Exclusive ORs the bit strings $\mathbf{str}_1$ ... $\mathbf{str}_n$ and returns as value the resultant string. The operation is performed left to right: the result of the XOR (Exclusive OR) of $\mathbf{str}_1$ and $\mathbf{str}_2$ is XORed with $\mathbf{str}_3$, that result is XORed with $\mathbf{str}_4$, and so on. None of the argument strings is changed by the operation.

An error is indicated if the strings are not equal in length.

STANDARD VECTOR FUNCTIONS

A Tabular Index to Primitive Vector Functions

Type of Vector	Predicate	Value of ELT	Specific Selection Function	Specific Updating Function	Allocating Function
Reference	REFVECP	Anything	ELT	SETELT	GETREFV
Selector Structure	SSVECP	Anything	ELT	SETELT	GETSELS
Character	STRINGP	Identifier	FETCHCHAR	STORECHAR	GETSTR
Bit	BITSTRINGP	Truth Value	ELT	SETELT	GETBITSTR
Word	WORDVECP	Integer	ELT	SETELT	GETWORDV
Real	REALVECP	Real	ELT	SETELT	GETREALV

Analogous functions for pairs are listed below for comparison.

Pair	PAIRP		CAR CDR	RPLACA RPLACD	CONS
List	LISTP	Anything	ELT	SETELT	LIST

Figure 7

(ELT **object index**)

This is the general selection function for vectors and lists. Its value is the **index** element of **object**, where the type of object returned by ELT is indicated in Figure 7 for various types of **objects**.

If **index** is not within the bounds of **object**, an error is indicated. If **object** is not a vector or a list, an error is indicated.

The first element of a vector or a list is selected by using zero as the **index** value. Note: ELT applied to NIL will always produce a bounds error, as NIL is interpreted as the empty list.

If **object** is a vector of reals, ELT will allocate a new real number into which the value of the selected element is copied, and return this new real to the caller. This is necessary (although not very efficient) because pointers pointing inside a vector are not allowed (they confuse the garbage collector). Thus, if a collection of real numbers are to be assembled into a vector, it is better to have a reference vector when access to these reals is made on an individual basis using ELT. The vector of reals exists for applications where the user has implemented arithmetic processes requiring the contiguous storage of real data in order to execute efficiently.

If **object** is a word vector, ELT may have to build a new large integer and return it as the value of ELT for certain values in the word vector. Any value within the range of a LISP small integer will be returned as a small integer, and will not require allocation of heap space. Values outside the range of small integers must be converted by ELT into large integers.

(GETREALV **n**)

Allocates and returns as value a real vector containing **n** elements. Each of the floating point values (elements) are initialized to zero.

(GETREFV **n**)

Returns as value a new reference vector containing **n** elements. The initial value of each element is NIL. This is the basic allocating function for reference vectors.

(GETWORDV **n**)

Allocates and returns as value a word vector containing **n** elements. The elements of the allocated vector are not initialized. See also GETZEROVEC.

(GETZEROVEC **n**)

Like GETWORDVEC, except the elements of the word vector are initialized to zero.

(LENGTHCODE **x**)

LENGTHCODE returns as its value the size, in bytes, of the vector **x**. If **x** is a character or bit vector, this value is the *maximum* size specified, including the 3-byte current string length field (see Figures 5 and 6)

An error is indicated if **x** is not a vector.

(LIST2FLTVEC **list**)

This function is similar to LIST2REFVEC but for the fact that the resulting vector is a vector of floating point (real) numbers. The elements of **list** may or may not already be real numbers. If they are not real, they are floated. If any element of **list** is not a number, or cannot be converted into a floating point number, the FLOAT function which is called by LIST2FLTVEC will take an error break.

(LIST2IVEC **list**)

This function is similar to LIST2REFVEC but for the facts that its value is an integer vector and the elements of **list** must be integers within the range acceptable for integer (word) vectors. This range is $2^{31} >$ value $\geq -2^{31}$.

(LIST2REFVEC **list**)

This function constructs a new reference vector from the elements of **list**. If **list** is non-pair, the value of LIST2REFVEC is a reference vector with zero elements. If **list** is a circular list, the function loops. Otherwise, the value is a reference vector of the form:

<(CAR **list**) (CADR **list**) ... (CAD...R **list**)>

(MAXINDEX **vector**)

Returns as value the maximum allowed index for the given vector. This function is defined as (SUB1 (SIZE **vector**)), so while its principal use concerns vectors, it could be applied to other objects as well. See also LENGTHCODE, LENGTH, SIZE.

(MOVEVEC **to from**)

Copies the contents of vector **from** into vector **to**. Both arguments must have the same capacity, and they must be both reference vectors or both binary (character, bit, real or word) vectors.

(REFVECP **x**)

Returns **x** if it is a reference vector, else returns NIL.

(SETELT **obj index value**)

This is the inverse function of ELT -- it updates the **index** element of **obj** to be **value**. The nature of **value** for various types of **obj** is indicated in Figure 7.

SETELT will take an error break if **obj** is not updatable, if **index** is out of range, or if **value** is not compatible with the type of **obj**.

SETELT may be used to update the **index** element of a list. For example,

(SETELT **list** 3 **value**)

is equivalent to:

(RPLACA (CDR (CDR (CDR **list**))) **value**)

The value of SETELT is the last argument for SETELT, the value to be used in updating the specified object.

(SIZE **x**)

The size function is the analog for vector arguments of LENGTH which applies to list arguments. SIZE returns as its value the *current* number of elements in its argument -- that is, one more than the maximum valid index which may be used to address an element of this vector. SIZE may also be applied to lists, in which case it performs exactly as does LENGTH. If SIZE is given an argument other than a pair or vector, it returns the value zero.

Note that string vectors (either character or bit) may have a capacity larger than the number of elements currently in the vector. See also LENGTH, LENGTHCODE, MAXINDEX.

(VECP x)

This function returns the value x if x is any variety of vector; otherwise, it returns the value NIL. This is a basic function, therefore it is not redefinable.

SUPERVISORY FUNCTIONS

(,SETGLOFN **x**)

This function is used to update the *glonot* component of the current global environment. The value of **x** should be either NIL or a pair. The value of ,SETGLOFN is the value of *glonot* before replacement.

The *glonot* component of a global environment is originally specified by optional arguments to the STATE function which created the state having that global environment. There are two such arguments, termed *gloval* and *gloalo*. When a binding is sought for a variable which has no binding in the current environment, *gloval* is applied to the identifier for which a binding is sought (first argument) and the *glolst* (global environment binding structure, typically an association list) of the current global environment. The value of this application will become the value of the binding to be created for this variable. It is permissible for this application to have other effects, such as generating an error break, et cetera. If *gloval* is NIL, the default action of assuming the identifier itself to be the binding value will be taken.

Once a value is available, *gloalo* is applied to the binding value, the identifier, and the *glolst* of the current global environment. The value of this application must be a pair (id . value) defining the new binding. Optionally, *gloalo* may have side effects such as updating the *glolst* structure. If *gloalo* is NIL, the default action of adding another element to *glolst* (assumed to be an association list) is taken.

The value of **x**, the argument of ,SETGLOFN, is the pair (*gloval* . *gloalo*). If **x** is NIL, the system default action will be taken.

(ADDOPTIONS **option**$_1$ **value**$_1$ ... **option**$_n$ **value**$_n$)

This function augments the free variable OPTIONLIST by APPENDing ((**option**$_1$. **value**$_1$) ... (**option**$_n$. **value**$_n$)) to the current OPTIONLIST. OPTIONLIST is an association list referenced by various functions such as DEFINE and COMP370, and is discussed in the section titled *Define, Compile and Assemble.*

For example, to set the NOLINK flag true, one might use:

```
(ADDOPTIONS (QUOTE NOLINK) 1)
```

(AERROR **x**)

This function is simply the compiled expansion of the macro (ERROR **x**). It is useful when it is necessary or convenient to apply a normal function (instead of a macro) in order to effect an error break. This happens most frequently in LAP-coded functions, because the compiler is very happy to expand the ERROR macro and compile the

resulting code, making it easy to change the definition of ERROR by simply recompiling. If, however, the expanded code were included in a LAP program, it would not be sufficient to reassemble that program with a changed ERROR macro definition because LAP does not expand such macros.

Also see AERRORR, ERROR and ERRORR.

(AERRORR x)

This function is simply the compiled expansion of the macro (ERRORR x). See the description of AERROR for a discussion of its use, and ERRORR and ERROR.

(BOUNDP **id**)

If there exists a binding for the identifier **id** anywhere in the current environment (including the current global environment), BOUNDP returns **id** as its value. If there is no such binding, BOUNDP returns NIL. No binding will be created for **id** by executing this function.

(ED $name_1$ ... $name_n$)

This is a macro which constructs a character string from the arguments $name_1$, ... , $name_n$ and invokes OBEY to pass control from LISP to a context editor. The character string created is:

'E $name_1$ ... $name_n$'

where $name_1$, etc. are exactly as provided as arguments to the macro (they are not evaluated). ED is thus intended to be used from the top level of LISP, where it is typed by the user.

Which context editor is invoked is controlled by the user, since he may use a variety of facilities to define exactly what program or command procedure is invoked under the name E.

(EMBED **id form**)

This function replaces the current value of **id** with a new value consisting of a lambda expression which has a modified **form** as its body. The modification consists of substituting the original value of **id** wherever **id** appears in FORM.

Also see MONITOR, which uses EMBED in a standard way to implement a TRACE facility.

Example: if FOO is a function of two arguments and it is desired to (temporarily, in lieu of redefining FOO) print the value of FOO and return NIL as its value regardless of its arguments, the following may be done:

```
(EMBED "FOO "(LAMBDA (X Y) (SEQ (PRINT (FOO X Y)) (EXIT ()))))
```

The value of EMBED is the identifier **id**. If **id** is not an identifier, an error will be detected when EMBED tries to do a SET.

(EMBEDDED)

The value of this function is a list of all **id**s under control of EMBED.

(ERROR ...)

This macro causes a channel 14 error break. It may have an indefinite number of arguments, which are put into a list which is bound to the variable ?ARGS?, which is available in the environment of the error break to define the particular nature of the error. Error channel 14 does not allow the invoking function to be continued, but requires an UNWIND. Also see ERRORR, which permits continuation from the point of invoking the error break, and AERROR and AERRORR.

ERRORINSTREAM

A fluid variable bound to a console input stream and used by the standard error break. It may be bound to a different stream by the user if it is desired to vary the operation of the error break.

Also see ERROROUTSTREAM.

(ERRORN ...)

This is a function of an indefinite number of arguments which is present for compatibility with the earlier LISP/360 system. It builds a character string by concatenating the string representations (see STRINGIMAGE) of the several arguments into one string which is used as an argument for ERROR.

ERROROUTSTREAM

A fluid variable bound to a console output stream and used by the standard error break code. It may be bound to a different stream by the user if it is desired to modify the action of the error break.

Also see ERRORINSTREAM.

(ERRORR ...)

This function causes a channel 12 error break. There may be an indefinite number of arguments supplied, which are put into a list which is bound to the variable ?ARGS? in the environment of the error break. This variable can be examined from the break loop to secure a precise idea of the nature of cause of the error. Error channel 12 permits continuation of the invoking function. This is achieved by invoking FIN. The first argument of FIN (or NIL if none is specified) is evaluated and becomes the value of ERRORR.

(ERROR2 ...)

A macro which expands into (ERRORN ...). It is present for compatibility with LISP/360.

(ERROR3 ...)

A macro which expands into (ERRORN ...). It is present for compatibility with LISP/360.

(ERR2 **channel-number**)

A macro which expands into an expression suitable for invoking the specified error channel. Correct usage involves applying this value to an expression describing the details of this error, which will be bound to the variable ?ARGS?. For example, to invoke error channel 12, one might write:

```
((ERR2 12) 'Sample of channel 12 error')
```

(ERR4 **channel sd** ...)

Part of the error break machinery. **channel** is used to look up (via ASSQ) an appropriate error channel in the fluid variable PROGRAM-EVENTS. This normally produces a list (**channel** errorfunction . channelid). In this case, errorfunction is applied to the three arguments **channel**, **message** and **sd**, where **message** is the list (...) containing all of the trailing arguments of ERR4.

In the atypical case where the requested **channel** cannot be found in PROGRAM-EVENTS, the standard errorfunction S,ERRORLOOP is used in lieu of an errorfunction obtained from PROGRAM-EVENTS.

(EXF **instream outstream options** ...)

This function is intended as a convenience for the interactive user wishing to process disk files using SUPV. After processing **options** ..., EXF will invoke SUPV:

```
(SUPV (DEFIOSTREAM ...) (DEFIOSTREAM ...))
```

where the arguments of the calls to DEFIOSTREAM are prepared using the disk file identification information given by **instream** and **outstream**.

If **outstream** is not specified, or is specified as NIL, the current value of CUROUTSTREAM (which is initially a console output stream) will be used for **outstream**. If **instream** is not specified, or is specified as NIL, the current value of CURINSTREAM (which is initially a console input stream) will be used for **instream**.

EXF is defined by a LAM so that it receives an indefinite number of unevaluated arguments. This is deemed appropriate for its intended use by an interactive terminal user.

instream and **outstream** may be specified as either a single identifier, which will be taken as the primary disk file name component, or as a list of file name components. If **outstream** is specified as the character object =, then an output disk file is defined using the filename of the input file, and a filetype of EXF. If **instream** is defaulted, and an output filename is required but not explicitly specified, the current value of CURINSTREAM will be used as the filename.

For example, if there exists a file IN1 LISP370,

```
(EXF IN1)
```

will invoke SUPV as follows:

```
(SUPV (DEFIOSTREAM (QUOTE ((FILE IN1 LISP370)))
        0 1) CUROUTSTREAM)
```

Another example:

```
(EXF (IN2 LISP D1) =))
```

will invoke SUPV as follows:

```
(SUPV (DEFIOSTREAM (QUOTE ((FILE IN2 LISP D1)))
      0 1)  (DEFIOSTREAM (QUOTE ((FILE IN2 EXF)))
              (CALLBELOW (QUOTE TOULL)) 1)
```

where (CALLBELOW (QUOTE TOULL)) yields the current terminal output line length.

If any options are specified, there should be an even number of arguments. In general, EXF will use the ADDOPTIONS function to augment the current value of the fluid variable OPTIONLIST, which contains various data directing the operation of such functions as the LISP compiler and LAP (see the section titled DEFINE, COMPILE and ASSEMBLE for OPTIONLIST values specifically affecting these functions).

In addition to the option values specifically coded by the user, EXF adds a PUSH option to OPTIONLIST, whose value is the current OPTIONLIST before it is augmented by EXF. This value is used after SUPV has completed to restore OPTIONLIST to the value it had when EXF was entered.

Several option values also have special meaning for EXF. If an output disk file is being produced, and the value of CUROUTSTREAM when EXF is entered is a console stream, then EXF adds the MESSAGE option to OPTIONLIST, where the value of the MESSAGE option is this console stream. The purpose of this is to make any error or warning messages generated by the compiler or LAP appear on the user's console as well as in the output

If the value of the FILE option is the character object = (after any explicitly written options have been added to OPTIONLIST by EXF), then a disk output stream is defined by EXF with fixed-format records of length 80 and a fileid composed of the filename from **instream** and a filetype of LISPLIB. Then a new FILE option whose value is this stream is added to OPTIONLIST.

If an output disk file is specified by **outstream**, the characteristics of that file may be set in several ways. In the absence of any other specification, the record format is defaulted (i.e. whatever DEFIOSTREAM does is what you get) and record length is obtained as indicated by the example above for console output. If there is an OUTPUTLENGTH property on OPTIONLIST, whose value is a small integer (after **options** ... have been added to OPTIONLIST by EXF), then this becomes the record length of the output file. A FILEDESCRIPTOR property is looked for on the property list of the identifier designating the filetype of the output file. The value of this property, if it exists, should be a list whose first two elements are record format (F or V) and record length, respectively. If a FILEDESCRIPTOR property is found, its record format specification is used for **outstream** and its record length specification is used if there is no OUTPUTLENGTH value on OPTIONLIST.

Different users have individual preferences in the matter of EXF, and a variety of similar functions have been written. If this version of EXF is displeasing, seek out one of these alternatives, or write something tailored to your personal tastes, or, if this version of EXF is not too far from what you want, simply modify it to do what you want (and please call it something other than EXF if users of your system are likely to get into trouble should they expect it to perform as described here).

EXTERNAL-EVENTS-CHANNELS

Fluid variable serving to parameterize the operation of the external interrupt processing program, DISPATCHER. It is bound to a reference vector containing the names

of the proper functions to be invoked for the corresponding external interrupt channel.

(EXTERNAL-INTERRUPT)

The standard function used to process an external interrupt produced by intervention of the user from his console.

In the case of VM, this is done through use of the CP command EXT which generates an external interrupt for the user's virtual machine. When LISP was loaded, it established an interface with CMS to obtain control in the event of such an interrupt, and the LISP system-dependent routine which receives control posts a LISP control structure indicating there is an interrupt pending which must be synchronized with the operation of the LISP system. The next time a LISP function executes an interrupt poll, control passes to the DISPATCHER program, which scans the LISP external interrupt control structure to determine which interrupt channel requires service, plucks the name of the corresponding service program from the EXTERNAL-EVENTS-CHANNELS vector and invokes it.

(FILELISP **fileid**)

Performs a garbage collection, then uses a system-dependent routine to write a complete copy of the current LISP system into an external file identified by **fileid**. If this file already exists, it is replaced by the new file.

The file image produced by FILELISP may be subsequently loaded by the LISPGET program (not a LISP program, but the program which loads and initializes the LISP system), whereupon the effect in the newly loaded system is of a return from the FILELISP function with a small integer value denoting the number of times in the history of this particular LISP system that a FILELISP has been performed.

FILELISP does not imply the abandoning of a running LISP system. After it has completed the writing of a file image onto external storage, it returns to its caller with a value which is a list of the **fileid** components (in the form of character strings) which were used to write the file image.

Example:

```
(FILELISP "(TEST FILEIM A1))
```

writes an image of the current LISP system into the designated file, then returns the value

```
('TEST    ' 'FILEIM  ' %9'A1')
```

There are several things which may go wrong with the FILELISP operation, ranging from I/O errors to missing parameters. If an error is detected, a channel 15 (LISP machine check) error break is taken and the variable ?ARGS? will be bound to some explanatory message.

(FIN **value**)

Used to exit from a LISP supervisor. The argument is optional, and a value of NIL will be assumed if no explicit value is provided. If the top-level supervisor encounters a FIN, it will leave LISP and return control to the program invoking LISP. Other uses of FIN involve nested invocations of the LISP supervisor SUPV, for example while using EXF.

Another use is in an error break loop, where control may be returned to the caller of the function entering the error break and **value** then appears as the value of the called function which entered the error break loop. To illustrate this latter use, assume the following expression is being evaluated:

```
(SETQ FOO (PLUS 2 '9'))
```

This is obviously invalid, as PLUS will not be able to cope with the character string argument. Thus an error break will be entered from PLUS, from which the user may exit with (FIN 11), which will supply 11 as the value of the PLUS expression which becomes the value of FOO.

(FUNARGSTATE **funarg**)

Returns as its value the environment (saved state) of its argument. If the argument value is not a *FUNARG*, an error break occurs.

(GCMSG **code**)

This function sets or resets the bits controlling typing of messages on the user's console after each LISP garbage collection. If **code** is NIL or 0, no messages are typed. If **code** is the identifier IND or 2, then data from a CP IND USER * command is typed (when running under VM/370). If **code** is 1 or anything else, a message indicating date and time, amount of heap and stack space before and after garbage collection, and cumulative CPU time is typed.

The value of GCMSG is a small integer indicating the argument value used for the previous call to GCMSG. It is possible to use this value to restore the previous condition when a temporary change is to be made.

(NILBOUNDP x)

This function returns true if there exists a NIL-environment binding for the identifier **x**. If there is no NIL-environment binding for **x**, NILBOUNDP returns NIL, without creating any NIL-environment binding. If **x** is a gensym, or is not an identifier, value of NILBOUNDP is NIL.

(NILSD)

This function returns as its value a state descriptor for the initial LISP state, before any bindings were made on the stack. Bindings made in the global environment are present, of course. One use of this state is to achieve what used to be called a "big unwind". For example, application of this state will pass control to HIGHLORD, the function which starts the LISP supervisor:

```
((NILSD) ())
```

Another use of this state is as an argument to EVAL, when it is desirable to perform an evaluation which doesn't see any current LAMBDA-bindings. Thus,

```
(EVAL EXP (NILSD))
```

computes the value of EXP in the global environment.

(JAUNT **sd value**)

This is a macro which expands into the expression (**sd value**). **value** is optional, and if not specified, the value NIL is used as a default. Other than this optional default value, the only purpose of the macro is to provide a specific, mnemonic indication that the user intends to jump to a new state without allowing a return to the calling function. See also STATE and EVAL.

(LERR4 **message channel sd**)

This is a binary program which invokes the specified error channel. Normally, the macro ERR4 (which uses a different order of arguments) would be used by the LISP programmer. LERR4 exists because a system programmer writing a LAP program has no macro expansion facility capable of dealing with the ERR4 macro, but may write code which invokes LERR4. See also LERROR.

(LOADVOL **fn ft fm**)

This function loads an entire lisp library file into the LISP/370 system. **fn** is the CMS file name, **ft** is the file type, and **fm** the file mode of the library file to be loaded. **fm**

and **ft** are optional. The default values are **fm=*** and **ft=**LISPLIB. All argument values should be identifiers.

LISP library files are most commonly constructed in toto from source files through use of the EXF macro. The process of loading items from a library file is essentially a series of assignments of values read from the file (a special reader is used which is very efficient because it doesn't have to parse symbolic expressions) to variables which are to have these values. The names of these variables are also recorded in the library file.

For detailed specification of the format of library file records, see the initial DSECTs in the LISP/370 assembly listing.

See also FETCH, RDEFIOSTREAM, FILEQ and SETANDFILEQ.

(OBEY x)

OBEY passes the character string **x** to CMS, or to CP, for execution as a command. The value of OBEY is the LISP small integer equivalent of the value returned by CMS or CP in register 15.

Any CMS or CP command may be set up by the character string. Moreover, the command may be the name of some EXEC procedure.

Examples:

```
(OBEY 'Q TIME')
(OBEY 'SET IMPEX OFF')
(OBEY 'LISTFILE * SCRIPT A1')
```

Note that abbreviations and synonyms are valid for both EXEC procedures and CMS/CP commands. The search hierarchy for file names is the same as defined for CMS.

These CMS/CP return codes are of special interest:

```
 0 : successful completion of the command
 1 : not EXEC procedure or CMS or CP command
-3 : not EXEC procedure or CMS command, and
     IMPCP control set OFF
45 : invalid option on command, or
     command abbreviated and ABBREV control set OFF
801: explicit EXEC with abbreviated file name when no
     USER synonym table in effect
```

(OUT-OF-HEAP)

Function invoked when there is insufficient space left in heap after a garbage collection. The motivation here is to allow the user to redefine this variable so that such a

condition would invoke a function to perform an UNWIND, or perhaps operate on a data structure to create more space by deleting some data. The original definition for this function explicitly invokes an error break.

(OUT-OF-STACK)

Function invoked when there is insufficient space left for stack frame after a garbage collection. The motivation here is to allow the user to redefine this variable so that such a condition would invoke a function to perform an UNWIND, or perhaps operate on a data structure to create more space by deleting some data. The original definition for this function explicitly invokes an error break.

(POST **k data**)

This function sets an interrupt pending in external events channel **k**. The LISP/370 system will not allow LISP interrupts to be processed at arbitrary times, but only when it is safe to do so. Thus, a poll is made for pending interrupts on entry to most functions (the only exceptions are some system functions which are non-interruptible because of their esoteric character, such as the garbage collector), and whenever an upward branch is to be executed, to afford an opportunity to interrupt program loops. Machine interrupts which do not require interruption of the normal sequencing of LISP programs, such as I/O interrupts and the like, are invisible to LISP and are controlled by programs external to the LISP system.

The **data** argument to POST may be any LISP expression, whose value is to be associated with the interrupt set by this call to POST. The definition of what actions are to be taken to service this interrupt are contained in the structure bound to the variable EXTERNAL-EVENTS-CHANNELS. In servicing an interrupt, the routine DISPATCHER applies the value of the proper channel's expression in EXTERNAL-EVENTS-CHANNELS to the value of POST's **data** argument.

This takes care of asynchronous interrupts, which may be posted not only by the routine POST, but by other processes external to the LISP system if suitable code is written by the user. LISP uses a special purpose data structure for queuing external interrupts, and the code for POST provides an example of how to find it and how it may be used.

There is another data structure analogous to EXTERNAL-EVENTS-CHANNELS, bound to the variable PROGRAM-EVENTS, which handles synchronous interrupts. These interrupts are invoked explicitly by LISP programs, so no queuing or provision for interaction with external processes is necessary. Functions such as ERROR make use of this latter mechanism.

(PROG *bv* e_1 ... e_n)

This macro implements the traditional LISP prog expression by expanding into

((LAMBDA *bv* (SEQ e_1 ... e_n)) () ...)

(PROGN e_1 ... e_n)

A macro which expands into (SEQ e_1 ... e_n).

(PROG2 e_1 e_2)

A macro which expands into (SEQ e_1 (EXIT e_2)).

(PRY ...)

A function which intentionally causes a program interrupt in order to enter a debugging sub-system. Execution of LISP may be continued by loading the old PSW stored at the time of the interrupt.

(RECLAIM)

This function explicitly invokes the LISP garbage collector. Its value is a reference vector which contains as its third and fourth elements (indices 2 and 3 because of zero-origin indexing) the number of bytes of heap space and stack space remaining after garbage collection. See also GCMSG.

(RET x y)

A function for passing control back to the caller of LISP, in a manner such that LISP may be reentered and resume execution following the call to RET. Both arguments are optional. x, if specified, must be a small integer which designates the return code (value in register 15) which will be provided to the caller of LISP. y is meaningful only if LISP has been invoked directly by means of a branch from another program instead of normally through the command interpreter. In the former situation, y may specify any LISP expression, and the pointer value of y will be returned to LISP's caller in register 1.

RET is most commonly used after LISP has been invoked to return to command level, whence a subsequent STRTLISP command will resume execution in LISP.

(SEQ e_1 ... e_n)

This is a special form in LISP/370 which establishes a statement context and denotes that the expressions e_1, ..., e_n are to be evaluated in sequence. If any of the e_i are identifiers, they are taken as labels which may be the arguments of a GO statement. It is possible to nest sequences within sequences, however it is not possible to GO from one sequence to a label defined in an internal sequence. It is possible to GO from an internal sequence to a surrounding sequence, provided the sequences are directly nested and no intervening expression context exists. For example,

(SEQ ... A ... (SEQ ... (GO A) ...))

is fine, but

(SEQ ... A ... ((LAMBDA () (SEQ ... (GO A) ...))))

is not valid, for the lambda expression raises a contour between the GO statement and its target label.

Frequently, a SEQ is used in a COND expression. For example,

```
(COND (MUMBLETYPEG
          (SEQ (PRINT MUM) (PRINT PEG) (PRINT N)) ))
```

(SETFUZZ (CONS **fuzz ndigits**))

Use this function to set or examine the values of **fuzz** (which specifies the desired precision of floating point operations) and **ndigits** (which specifies the criterion used by the print routine to determine whether to print a real number with or without an explicit exponent).

SETFUZZ takes one argument, which is a pair whose CAR component is a positive real number which will become the new value of **fuzz**, and whose CDR component is a positive integer which will become the new value of **ndigits**. The value of SETFUZZ is a similar pair containing the previous values of **fuzz** and **ndigits**, so that

(SETFUZZ (SETFUZZ (CONS **fuzz ndigits**)))

restores the original values of **fuzz** and **ndigits**.

The use of the values of **fuzz** and **ndigits** is discussed in the section on data types.

(STATE [[**glonot**] **glolst**])

Saves the current state or a modified form of it in the case that optional arguments were supplied. The modified form of the current state may differ only in the global environment ***gloE*** component of the environment ***E***. The value is a state descriptor *sd*

which denotes the state. This *sd* may be used as an argument to EVAL to provide the environment of the above state as the bindings context for the evaluation. The *sd* may be applied to a message argument causing the saved state to continue. In that case the value of the STATE operator is not the *sd* but the message.

The optional arguments **gloval** and **glonot** describe the modifications to the ***gloE***.

A ***gloE*** is a special object with two components:

glonot the not present prescription for this ***gloE***

is a pair (*gloval* • *gloalo*) where

gloval is NIL or else a two argument function from the *id* in question and the *glolst* of the current ***gloE***, to the *s-exp* value for that variable in this global environment.

gloalo is NIL or a three argument function from *s-exp, id, and glolst* to *globnd* values. Often the side effect of updating *glolst* is accomplished.

and *glolst* the global data list structure environment is

({*glodat* | *globnd*} • {*glolst* | *glotrm*}) ,

and *globnd* the global binding is a *pair* (*id* • *s-exp*),

and *glodat* the global own data, is any *s-exp* which is not a *pair*,

and *glotrm* the global environment terminator is, {NIL | *sd*}.

(SUPERMAN)

This function is actually a continuation of the initialization performed by HIGHLORD. Fluid bindings for CUROUTSTREAM, et cetera, are made and SUPV is invoked.

(SUPV **instream outstream**)

This is the usual LISP supervisor for LISP/370. Its first argument is the input stream from which expressions will be read and evaluated. SUPV binds the fluid variable ,NEWSTATE to an appropriate state so as to set up for a subsequent (UNWIND) in the event of an error while reading an expression, evaluating it, or printing its value.

Two free variables control the prolixity of SUPV. If ,ECHOSW is true, then SUPV will print the expressions it reads on **outstream**. If ,VALUSW is true, SUPV will print the values of the expressions on **outstream**.

Before invoking READ, SUPV tries to get a character from the input stream. If a null line is encountered at this point, SUPV prints the message LISP on the output stream and tries again to get a character. Once SUPV has started reading an expression, control resides in the READ function and null lines will be ignored.

Upon detecting an end of stream condition for **instream**, SUPV returns with the value 1. Otherwise, SUPV will return only when it reads the expression (FIN exp), where exp is optional. If exp is specified, it will be evaluated and its value returned as the

value of SUPV. If exp is not specified, NIL is returned as the value of SUPV. In either case, when SUPV returns, INFILE and OUTFILE have been SHUT.

A typical use of SUPV to read and evaluate expressions from the disk file 'TEST LISP' is:

```
(SUPV   (DEFIOSTREAM (QUOTE ((FILE TEST LISP))) 0 1)
        CUROUTSTREAM)
```

where output is to be written onto the default output stream, which is initially LISPOT (a console output stream).

(SYSID)

This function returns a code indicating which operating system is running LISP/370. A value of 1 indicates VM/CMS, a value of 2 indicates MVS/TSO.

SYSID is used by some standard functions such as ERASE and IOSTATE, which are implemented by various calls to the OBEY function, depending upon the underlying operating system. SYSID may similarly be used by user functions which which to maintain some degree of operating system independence by tailoring their action according to the operating system which is running LISP.

(UNEMBED **id**)

Undoes the effect of a previous EMBED for **id** in the current environment. The value of UNEMBED is the identifier **id** if **id** was previously EMBEDDED. If not previously EMBEDDED, the value of UNEMBED is NIL.

STANDARD I/O FUNCTIONS

(,FILEIN **stream**)

Stream specific function for fast disk input streams.

(,FILEOUT **x stream**)

Stream specific function for fast disk output streams.

(CONVERSATIONAL)

This function uses a system-dependent portal to determine whether LISP is running with an on-line, connected terminal (at which there is presumably a user ready to interact with the LISP system), or whether LISP is running in batch mode or without a connected terminal. The value of CONVERSATIONAL is true if an interactive terminal is present, otherwise the value is NIL.

CURINSTREAM

This fluid variable is bound by SUPV to the current input stream. It is used by the READ macro to provide the necessary value for the stream from which an expression is to be read when an explicit value is not provided by the invocation of READ.

CUROUTSTREAM

This fluid variable is bound by SUPV to the current output stream. It is used by the various PRINT macros to provide the necessary stream value when an output stream is not explicitly provided by the invocation of PRINT.

(CURRINDEX **x**)

If **x** is a fast stream (the type of stream generated by DEFIOSTREAM), the current index is returned as the value of CURRINDEX. If **x** is not a fast stream, the value of CURRINDEX is NIL.

(DEFIOSTREAM **alist buffersize itemnumber**)

This function constructs a standard structure usable as an input/output stream by the normal READ and PRINT programs. It is not intended to be capable of building all

possible streams, merely those which are most commonly used. The structure which is created and returned as the value of DEFIOSTREAM is not checked for validity by DEFIOSTREAM. The first use of the stream will typically involve initialization and verification of the data in the structure according to the needs of the using function.

For INPUT mode the structure created by DEFIOSTREAM has the following general structure:

```
%L1=  (%L1 . < rfn
                < () 0 buffersize buffersize >
                alist
                ()
                itemnumber> )
```

For OUTPUT mode the structure created is:

```
%L1=  (%L1 . < rfn
                < %n'...' 0 0 buffersize >
                alist
                ()
                itemnumber> )
```

alist is an association list which defines some of the characteristics of the stream being created. **buffersize** is an integer defining the length of the lines for which buffer space is to be provided. This represents a maximum length; shorter lines may be produced by using TERPRI for output, and shorter lines may be emitted by whatever source an input stream uses. A line buffer is not allocated by DEFIOSTREAM, but will be allocated the first time the stream is used by READ or PRINT. **itemnumber** designates a particular record within a data set. A value of zero means use the first record if an input stream, or the next record if an output stream.

Streams may be defined by this function based either on a disk file or console as an input/output device. The **alist** value is examined to determine which of these devices is to be used, and an appropriate program is selected and stored as the value of rfn in the stream structure.

In order to implement commonly-needed default stream attributes, DEFIOSTREAM makes the following modifications to **alist**:

> If no MODE property is already part of **alist**, (MODE . INPUT) is added to **alist** by non-destructive CONSing.
>
> If (MODE . I) or (MODE . O) is specified, I or O is RPLACDed by INPUT or OUTPUT, respectively.

Following is a description of the properties and meanings which are most commonly used for **alist**. Additional properties are ignored by the standard stream processing

functions, but may be added to provide additional information to be used by the user's programs.

(DEVICE . CONSOLE) is specified to indicate this stream is defined on the user's console.

(FILE filename filetype) or (FILE filename filetype filemode) is specified to indicate this stream is defined on the designated disk file. Values for filename and filetype may be specified either as identifiers or character strings.

(MODE . INPUT) or (MODE . OUTPUT) designates an input or output stream, respectively. (This attribute was discussed above.)

(RECFM . V) or (RECFM . F) designates whether a disk file referenced in an output stream is to have fixed or varying length records in it. Varying length records is the default assumption.

(QUAL . S or T or U or V or X) designates the type of CMS read operation to be used in obtaining records from the console for this stream. The letters have the following meanings:

S = pad records with blanks to 120 characters.
T = read a logical line (the default operation).
U = pad with blanks and translate to upper case.
V = translate to upper case.
X = read a physical line.

(QUAL . LIFO or FIFO or NOEDIT) specifies for console output files that no editing is to be performed on output lines (i.e. for typewriter consoles, trailing blanks are not deleted and a carriage return is not automatically appended to the line). LIFO and FIFO designate that output lines are to be placed into the console input stack, rather than be written to the console.

(DIGIT **x**)

Returns **x** if it is one of the character objects (identifiers) 0, 1, 2, 3, 4, 5 ,6 ,7, 8 or 9. Otherwise, returns NIL. This is a macro which expands into in-line code for compiled programs.

(DOMINATESTREAM **topstream bottomstream**)

This function requires two fast streams as arguments, and causes each line which is subsequently written into the first stream to also be written into the second stream. **topstream** is termed the dominated stream, and **bottomstream** is said to be the dominating stream. This piece of prestidigitation is achieved by modifying the stream-specific

function of **topstream**, so that whenever it would have been invoked, the modified function is used which copies the **topstream** buffer into **bottomstream** and TERPRI's **bottomstream**, as well as performing the function of the original **topstream** stream-specific function. **bottomstream** is not modified by this process, and **topstream** may be restored to its original state by use of the function UNDOMINATESTREAM.

There are some limitations. First, as already indicated, both streams must be fast streams (in order that the buffer copying code may do the right thing). This concept of stream domination is applicable to slow streams also, but the function DOMINATESTREAM will not implement it. Second, the buffer size of the dominated stream, **topstream**, must be no larger than the buffer size of the dominating stream.

(ERASE **fname ftype [fmode]**) or

(ERASE (**fname ftype [fmode]**))

This function erases a file, or group of files, to which the user has write access. The value of the function is true if a file is successfully erased; otherwise, the value of the function is NIL.

fname and **ftype** must be specified. If **fmode** is omitted, the primary read/write disk A1 is the only disk searched. If both **fname** and **ftype** are specified by an *, **fmode** must be specified by other than an *. Note that the erasure of the entire A1 disk is not permitted, and care is advised whenever an asterisk is used as an argument so that files are not inadvertently erased.

ERASE constructs a character string containing a suitable command and OBEYs it. To provide a more similar operation with the ERASE function of an earlier LISP system, this character string is printed on CUROUTSTREAM.

(FASTSTREAMP **str**)

This function returns true if **str** is a fast stream -- that is, a pair whose CDR is a reference vector of length at least 3. Otherwise, the value of FASTSTREAMP is NIL.

(FILEQ *id* **value**)

Similar in intent to SETQ, except that instead of binding **value** to (the "quoted" i.e. unevaluated identifier) *id*, **value** is written into the LISPLIB currently accessible through the stream which is the value of the FILE property on OPTIONLIST. This provides a convenient method for writing more general objects into a LISPLIB than simply the proto-modules which are the output of LAP and indirectly of the LISP compiler.

There are some limitations on the types of objects which can be written into a LISPLIB. In effect, anything which cannot be printed in a form which can then be read back into the LISP system cannot be written into a LISPLIB. These are binary program images (not proto-modules, but the result of linking a proto-module into LISP's binary program space) and state descriptors, the value of the STATE function. It follows that any composite structure containing a pointer to one of these objects cannot be written into a LISPLIB.

Circular structures present no intrinsic problem, and may be written into LISPLIBs provided they do not contain any of the proscribed data objects.

FILEQ is of course a macro, in order to get a hold of *id* before it is evaluated. **value**, however, is evaluated to obtain the object to be written into the LISPLIB. Like SETQ, **value** is returned as the value of FILEQ.

(HEXEXP **x**)

Returns as value a character string containing the EBCDIC codes for the hexadecimal value of the pointer which is the value of **x**. For example,

(HEXEXP 10) = '0300000A'

(HEXNUM **x**)

Returns as value a character string containing the hexadecimal digits representing the value of **x**. For example,

(HEXNUM 10) = '0000000A'

The argument value, **x**, must be either a small integer or a single-word large integer, otherwise an error break is taken. See also HEXEXP.

(HEXSTRINGPART **string index length**)

Returns as value a new string which contains the hexadecimal character representation of the **length** characters of **string** beginning with the character selected by **index**. For example,

(HEXSTRINGPART 'ABCDE' 1 2) = 'C2C3'

(IOSTATE **fname** [**ftype** [**fmode**]]) or

(IOSTATE (**fname** [**ftype** [**fmode**]]))

This function returns the value true if a file exists with the name **ftype**, type **ftype** and mode **fmode**; otherwise, its value is NIL.

Only **fname** need be specified. If **ftype** is omitted, the first file encountered with the name **fname**, of any type, will result in the value true.

If **fmode** is omitted, all disks are searched. If **fname** is *, **ftype** or **fmode** must be specified for the search to be meaningful.

(IOSTATEW **fname** [**ftype** [**fmode**]]) or

(IOSTATEW (**fname** [**ftype** [**fmode**]]))

This function returns the value true if

1. a file exists with the name **fname**, type **ftype** and mode **fmode**, and
2. the user has write access to that file;

otherwise, the value of the function is NIL.

Only **fname** need be specified. If **ftype** is omitted, the first file encountered with the name **fname**, of any type, and to which the user has write access, will result in the value true.

If **fmode** is omitted, all disks are searched. If **fname** is *, **ftype** or **fmode** must be specified for the search to be meaningful.

(IS-CONSOLE **stream**)

This function returns NIL as its value if **stream** is not a fast console stream. If **stream** is such a stream, it is returned as the value of the function.

(ITEM-N-ADV **stream**)

Returns as value the current object at the head of the steam **stream**, then advances **stream** using NEXT. This function discards line end indications, so the caller will see only actual data values from **stream**.

(LISPITTIN **stream**)

Stream specific function for fast console input streams.

(LISPOTOUT **x stream**)

Stream specific function for fast console output streams.

(NEXT **stream**)

This is a basic function for manipulating streams. It advances **stream** to the next character and returns the updated **stream** as its value; the CAR of the **stream** is this next character. For more information, see the section on the LISP destructive stream facility. See also WRITE, TEREAD, TERPRI.

(NUD)

This is the principal function used by READ for parsing LISP symbolic expressions. Changes of syntax for the print representations of s-expressions usually involve changing this function, or one of its accessory functions.

(PRETTYPRINT **x stream**)

Similar to PRINT, except a more complicated program is invoked which understands many of the more common forms of symbolic expressions and prints them in a structured format. Unlike PRINT, PRETTYPRINT will not try to print structures with cycles in them. If a cycle exists in **x**, the regular PRINT function is invoked. PRETTYPRINT does not attempt to evidence shared structures as does PRINT, but prints each shared substructure in full.

The **stream** argument is optional. If it is not provided, the current value of the variable CUROUTSTREAM is used. PRETTYPRINT uses a free variable, PRETTYWIDTH, to define for it the maximum length of output lines.

(PRETTYPRIN0 **x stream**)

This is a subfunction of PRETTYPRINT which takes the same arguments (**stream** is optional), but does not perform a TERPRI after **x** has been printed.

(PRINM **x stream**)

A specialized print function which expects **x** to be a pair (otherwise it is CONSed with NIL and this pair is treated as **x**) and prints each element of the list **x**. There are no

top level parentheses printed, and if any element of the list **x** is a character string, it is printed by PRINTEXP instead of the normal PRIN0 routine, so that its delimiting characters are not printed. This function exists for purposes of compatibility with LISP/360, where it was used principally for printing error messages.

(PRINT x stream)

This is a macro which supplies the identifier CUROUTSTREAM if there is no explicit **stream** argument provided. It invokes the standard LISP/370 print routine, which writes the canonical output representation of the value of **x** into **stream**. This representation shows all shared substructure, including cyclical structure. The final operation of PRINT is to perform a TERPRI on **stream**. The value of PRINT is **x**.

(PRINTCH char stream)

This is a macro which makes optional the argument value for **stream**. If none is supplied, CUROUTSTREAM is used. The expansion of PRINTCH is

(WRITE **char stream**)

(PRINTEXP string stream)

This macro makes optional the argument value **stream**. If none is supplied, the value CUROUTSTREAM is used. The function ,PRINTEXP invoked by this macro requires **string** to be a character string, and writes this string into **stream** without string delimiting or letterizing characters.

(PRINTEXPPNAME string stream)

This function is used for the normal printing of print names of identifiers, where **string** is the print name to be printed. Necessary letterizing characters are inserted.

(PRINTVAL x stream)

Prints "VALUE =" then PRETTYPRINTs **x** into **stream**.

(PRIN0 x stream)

This macro is similar to PRINT, except the final TERPRI is not performed. **stream** is an optional argument, and CUROUTSTREAM will be used if no explicit value is supplied. The value of PRIN0 is **x**.

(PRIN1 **x stream**)

This macro makes optional the argument **stream**. If no value is explicitly supplied, CUROUTSTREAM will be used. The function invoked by this macro will print only non-descendable objects (i.e. no pairs or reference vectors). No blank is printed after **x**. The value of PRIN1 is **x**.

(PRIN1B **x stream**)

This macro is similar to PRIN1, but a blank character is written into **stream** after **x** is printed.

(PUTBACK **x stream**)

One of the functions for manipulating input streams. The value of **x** is pushed onto the head of the stream, where it becomes the current object at the head of the stream. This is meaningful only for input streams, and finds application in the READ programs, where a delimiter is encountered by an auxiliary function of NUD using ITEM-N-ADVANCE, and the delimiter is PUTBACK onto the stream where it will be subsequently seen by that part of the READ operation which is equipped to interpret it.

(RDCHR **stream**)

This is a macro which expands into an invocation of ITEM-N-ADVANCE, where the optional **stream**, if not explicitly specified, is given the value CURINSTREAM.

(RDEFIOSTREAM **alist**)

This function constructs a standard structure usable as an input/output stream by the RREAD and RWRITE programs. It is not intended to be capable of building all possible streams, merely that which is needed for simple random-access I/O in the LISP system. The structure which is created and returned as the value of RDEFIOSTREAM is not checked for validity, the first use of the stream will typically involve verification according to the needs of the using function.

The stream created has the following general structure:

```
%L1=  (%L1 . < rfn
                < %I<...> 0 4 80 >
                alist
                '...'
                position-key > )
```

(RDS **stream**)

This macro expands into (SETQ CURINSTREAM **stream**), and exists for compatibility with LISP/360 usage.

(READ **stream**)

This macro invokes the basic LISP symbolic expression reader. The **stream** argument is optional. If is is not explicitly supplied, CURINSTREAM is used.

READ references three free variables which control the parsing of input data. These are QUOTEIZER, STRINGIZER and LETTERIZER. Each should be bound to the character object which is to signal the READ program to perform an appropriate operation. The QUOTEIZER character indicates that the symbolic expression immediately following (there may be no intervening blanks before the first character of this expression) is to become the second element in a list whose first element is the identifier QUOTE. This is a particularly useful facility when typing expressions interactively from a terminal. The STRINGIZER character is simply the character designated to act as a string delimiter, and LETTERIZER is the character used to signal that the immediately following character is to be considered a data character rather than a control character.

The LETTERIZER character is needed for designating, for example, the identifier having the print name '999', or the character object left parenthesis or blank.

(READPLACEGEN)

Returns as value a unique read place holder. These objects are used by the READ programs to cope with input expressions which designate shared structure. See also PLACEP.

(RREAD **key stream**)

This function reads one record identified by **key** from the file specified by **stream**. **key** must be a character string. **stream** must have have been defined by RDEFIOSTREAM. If input is needed from a file written after **stream** was defined, the file must first be closed with RSHUT and the **stream** must be redefined. This is necessary in order to access the new data via the updated DIRECTORY of **keys**.

The value of RREAD is the record read.

(RSHUT **stream**)

SHUT, or close, the file specified by **stream**. The value of RSHUT is the updated **stream** looped to itself.

%L1=(%L1 . %L1)

(RWRITE **key item stream**)

Write one **item** identified by **key** into the file specified by **stream**. **key** must be a character string. Each **item** is written as an 80 character fixed-length record. Every write updates a DIRECTORY which is an association list of **keys** and record position numbers.

The value of RWRITE is **item**.

(SETANDFILEQ *id* **value**)

This macro is useful in source files to be processed by EXF or a similar mechanism where it is desired to both write **value** into a LISPLIB file as the value associated with the label *id* (which need not be quoted as the macro does not evaluate it), and to achieve a current assignment by performing a (SETQ *id* **value**). In effect, this is the combination of FILEQ and SETQ.

(SHUT **stream**)

This function invokes a system-dependent routine to close any file related to the argument **stream**. If no file is actually in need of closing, the action of SHUT is effectively a no-operation. The value of SHUT is the (possibly updated) **stream**.

(SKIP **n stream**)

This macro issues **n** TERPRI's to **stream**. If **stream** is not specified, CUROUT-STREAM is assumed. The value of SKIP is NIL.

(STRINGIMAGE **x**)

This function uses a specialized stream to accumulate a character string containing the print representation of **x**, which becomes the value of STRINGIMAGE. This may be useful in a variety of ways. For example, th obtain the character equivalent of an integer, one can write

(STRINGIMAGE **integer**)

x may be any LISP value.

(STRINGIZE **x**)

This function also computes a string representation for **x**. If **x** is a string, it is returned unchanged as the value of STRINGIZE. If **x** is not a list, the value of STRINGIZE is (STRINGIMAGE **x**). If **x** is a list, the value of STRINGIZE is the concatenation of the STRINGIMAGEs of the elements of **x**.

(TAB **n stream**)

This macro causes sufficient blanks to be written into **stream** so that the last character in the stream output buffer is a byte position **n**. If there are already more than **n** bytes in the current output buffer, a TERPRI is performed, and **n** blanks inserted into an empty output buffer.

(TEREAD **stream**)

This macro forces an end of line condition in the fast stream **stream**. Any characters left in the current input buffer are lost. **stream** is an optional argument. If it is not specified, CURINSTREAM is used.

(TERPRI **stream**)

This macro forces output of the current line in **stream**.

(UNDOMINATESTREAM **topstream**)

Removes the effect of the DOMINATESTREAM function. The argument must be the stream which was the **topstream** argument to DOMINATESTREAM, i.e. the dominated stream.

(WRS **stream**)

This macro expands into (SETQ CUROUTSTREAM **stream**), and exists for compatibility with LISP/360 usage.

STANDARD ARITHMETIC FUNCTIONS

(*MAX x y)

Returns as value the algebraically greater of the two numeric arguments x and y. This is a function of exactly two arguments. It is used by MAX, a macro which performs a similar operation for an indefinite number of arguments.

(*MIN x y)

Returns as value the lesser of the two numeric arguments x and y. This is a function of exactly two arguments. It is used by MIN, a macro which performs a similar operation for an indefinite number of arguments.

(ABSVAL x)

Returns the absolute value of x, if x is a number. If x is not a number, an error break is taken.

(ADD1 x)

Returns as its value (PLUS x 1). ADD1 is implemented as a macro which expands (for the compiler) into in-line code to compute the value if x and the result are both small integers, and calls a separate function (LPLUS) if one or both are not small integers. If x is not numeric, LPLUS will take an error break.

(ALINE n k)

Value is the small integer n rounded up to the nearest multiple of the small integer k, where k is a power of 2. If n or k are not small integers or are negative, an error break is taken. If n is zero or one, returns N unchanged. If k is not a power of 2, result is unpredictable..

Example: (ALINE 15 4) = 16.

(DIFFERENCE x y)

This is a macro which expands into the expression (LDIFFERENCE x y). LDIFFERENCE computes the numerical difference between two numbers. If any argument is real, the result is real. Otherwise, the result is an integer.

There is no strong reason for having a macro here. It does make for greater uniformity with PLUS, where the macro is needed to produce the expansion for more than two arguments, and may one day expand into different forms according to additional information defining the possible values of the arguments.

See also QSDIFFERENCE.

(DIVIDE x y)

This is a macro which expands into (LDIVIDE x y). LDIVIDE computes the quotient of x divided by y, and returns as value a list: (**quotient remainder**) . The first element of this list is the quotient and the second element is the remainder. If both x and y are integers, the quotient and remainder will be integers. If any argument is real, the quotient and remainder will be real. The remainder for a real quotient is peculiar because it exists only because of the approximation needed for representing real numbers as floating point numbers in the computer. A real remainder is computed as

(DIFFERENCE x (TIMES y **quotient**)).

(EQUALN x y)

The value of this function is the value of (EQUAL x y) when the floating point fuzz factor is zero. The fuzz factor is temporarily made zero while EQUAL is invoked, then restored to its original value.

(EXPT x y)

Returns the value of x raised to the y power. The value is an integer if x is an integer and y is a positive integer; otherwise, the value is a floating point number.

x cannot be negative if y is not a positive integer.

(FIX x)

If x is a real (floating point) number, returns the integral part of that value as an integer. If x is already an integer number, it is returned as the value of FIX; For other values of x, an error break is taken.

(FLOAT x)

If x is an integer number, the value is the closest real approximation to that number, unless x exceeds the range of floating point numbers (approximately the 80th power of

10), in which case an error break is taken. An error break occurs also if x is not numeric. If x is already a real number, it is returned as the value of FLOAT.

(FLOATP x)

This function returns the value x if x is a floating point number; otherwise, its value is NIL.

(GETFLT)

This macro expands into code which will allocate a new real (floating point) number cell when it is executed. It is used by functions such as EXP which need a new cell in which to return their value. The initial value in the new cell is zero.

(GREATERP x y)

For x and y numeric values, compares them and returns true if x is greater than y, or NIL if x is not greater than y. Should either x or y be a floating point value, the real fuzz factor may affect the comparison. There is a discussion of this in the section on data types. Basically, the fuzz factor allows two real values which are close in value to be considered equal, thus neither is greater than the other. The value of the real fuzz factor defines what close means. See also EQUAL, EQUALN.

(LDIFFERENCE x y)

This function computes the value x minus y, where x and y are numeric values. If either x or y is not numeric, an error break is taken. If both x and y are integers, the value of LDIFFERENCE will be an integer. Otherwise, the result value is real.

See also QSDIFFERENCE.

(LDIVIDE x y)

This function computes the quotient and remainder of x divided by y, and returns as value a two element list: **(quotient remainder)** . The first element of this list is the quotient, the second element is the remainder. If either x or y is not numeric, an error break is taken. If both x and y are integers, the quotient and remainder will be integers. Otherwise, the quotient and remainder will be real.

If the quotient is real, the remainder will be somewhat peculiar since it exists only as a result of the limited accuracy with which real numbers are stored in the computer. In this case, the remainder is computed as

(DIFFERENCE x (TIMES y **quotient**))

See also QUOTIENT.

(LEFTSHIFT **number count**)

If **number** is a small integer or one word large integer, the value of this function is the number (limited to the range of a one word large integer) obtained by a binary shift of **count** bits. Positive values of **count** denote a left shift, negative values a right shift. Any bits shifted outside of a 32 bit word are lost, and zero bits are supplied as needed. If **number** is not within the described range, an error break is taken. See also RIGHTSHIFT.

(LESSP x y)

This is a macro which expands into the expression (GREATERP y x).

(LN x)

Function for computing natural logarithm of x. x may be either integer or real, but it must be within the range of a real number. The result is a real number. See also LOG, LOG2.

(LOG x)

Computes common (base ten) logarithm of x. x may be either integer or real, but must be within the range of a real number. See also LN, LOG2.

(LOG2 x)

Computes the logarithm to the base 2 of x. x may be either integer or real, but must be within the range of a real number. See also LN, LOG.

(LPLUS x y)

This function computes the sum of x and y, where these arguments are numeric values. If either x or y is not numeric, an error break is taken. If both x and y are integers, the result is an integer. Otherwise, the result is real.

See also QSPLUS and PLUS.

(LTIMES x y)

This function computes the product of **x** and **y**, where these arguments are numeric values. If either **x** or **y** is not numeric, an error break is taken. If both **x** and **y** are integers, the result in an integer. Otherwise, the result is real.

See also QSTIMES and TIMES.

(MASKNUM **number nbits**)

When **number** is a small integer or a one word large integer, a new number is computed as the value of MASKNUM which consists of only the rightmost **nbits** of **number**, the remaining high-order bits being set to zero.

If **number** is not is the valid range, an error break occurs. See also LEFTSHIFT.

(MAX n_1 ...n_n)

This is a macro which expands into a nest of applications of *MAX. The value of this expression is the algebraically largest argument value. If any of the arguments is not a number, an error break occurs.

(MIN n_1 ... n_n)

A macro which expands into a nest of applications of *MIN. The value of this expression is the algebraically smallest argument value. If any of the arguments is not a number, an error break occurs.

(MINUS x)

Unary minus operation. **x** may be any number. Value returned is minus **x**.

(MINUSP x)

Returns NIL if **x** is positive or zero, true if **x** is negative.

(NUMBERP x)

This function returns the value **x** if **x** is any type of number. If **x** is not a number, the value NIL is returned.

(PLUS x_1 ... x_n)

This is a macro which expands into the expression

(LPLUS x_1 (LPLUS x_2 ... (LPLUS x_{n-1} x_n) ...))

LPLUS is a function of two numeric arguments which returns their sum as value. See also QSPLUS.

(QSDIFFERENCE x y)

This macro expands during compilation into in-line code to compute as its value the difference of x and y, under the assumption that x, y, and the result value are small integers. If interpreted, the macro expands into the expression (LDIFFERENCE x y). If the assumption of small integers is violated in compiled code, the result will be a small integer of incorrect value.

(QSPLUS x y)

This is a macro which expands during compilation into in-line code to compute as its value the sum of x and y, under the assumption that x, y, and the resulting sum are small integers. If interpreted, it is equivalent to (LPLUS x y). If the assumption of small integers is violated in compiled code, the result will be a small integer of incorrect value.

(QSTIMES x y)

This macro expands during compilation into in-line code which computes as its value the product of x and y, under the assumption that x, y and the result are all small integers. If interpreted, the macro is equivalent to (LTIMES x y). If the assumption of small integers is violated in compiled code, the result will ba a small integer of incorrect value.

(QUOTIENT x y)

This function is similar to LDIVIDE in that it computes the quotient of x divided by y, but differs in that it returns that quotient as a number rather than CONS'ing it into a list with the remainder of the division. If x and y are both integers, the quotient will be an integer. Otherwise, the quotient is real.

(REMAINDER x y)

Returns as value the remainder of x divided by y. If both x and y are integers, the remainder is computed from an integer division. If either x or y is floating point, the

remainder is computed by subtracting y times the real quotient from **x**.

(RIGHTSHIFT **x n**)

This macro expands into a call on LEFTSHIFT, where the sign of the shift amount, N is changed.

(SMINTP **x**)

This function returns **x** as its value if **x** is a small integer; otherwise, its value is NIL. This is a basic function, therefore it is not redefinable.

(SUB1 **x**)

Returns as its value (DIFFERENCE **x** 1). SUB1 is implemented as a macro which expands (for the compiler) into in-line code to compute the value if **x** and the result are both small integers, and calls a separate function (LDIFFERENCE) if one or both are not small integers. If **x** is not numeric, LDIFFERENCE will take an error break.

(TIMES $x_1 \ldots x_n$)

This is a macro which expands into the expression

(LTIMES x_1 (LTIMES x_2 ... (LTIMES x_{n-1} x_n) ...))

LTIMES is a function of two numeric arguments which computes as its value the product of those arguments. If all of the arguments of TIMES are integers, the value will also be an integer. Otherwise, the value of TIMES will be real.

See also QSTIMES.

PROPERTY LIST AND OBLIST FUNCTIONS

(ASSOC **item alist**)

This is the standard LISP ASSOC function.
alist is a list of pairs, ((**s-exp**$_1$. v_1) (**s-exp**$_2$. v_2) ...). ASSOC compares **item** with **s-exp**$_1$, then **s-exp**$_2$, ..., using EQUAL to perform the comparison. If **alist** is not a list, or if **item** is not found in **alist**, the value of ASSOC is NIL. Otherwise, the value of ASSOC is the first pair (**s-exp** . **v**) such that (EQUAL **item** (QUOTE **s-exp**)) is true. If there are elements of **alist** which are not pairs, they are skipped and the next element of **alist** is examined.

Also see SASSOC, ASSOCN, and ASSQ.

(ASSOCN **item alist**)

This function is very similar to ASSOC, except that EQUALN is used instead of EQUAL for comparing **item** with the CARs of elements in **alist**. Also see ASSQ and SASSOC.

(ASSQ **item alist**)

This function is similar to ASSOC, except it uses EQ rather than EQUAL to compare **item** with the CARs of elements in **alist**. Also see ASSOCN and SASSOC.

(DEFLIST **pairlist property**)

This function expects **pairlist** to be a list of pairs whose CARs are identifiers and whose CDRs are arbitrary values. For each of these pairs,
(MAKEPROP **id property value**) is performed, assigning the CDR value from the pair as the value of the **property** property in the identifier's property list.

The value of DEFLIST is a new list containing the property values (the CDRs of the elements of **pairlist**).

(GET **id propname**)

If **id** is not an identifier or a pair, value is NIL. Otherwise the property list of **id** (or **id** itself, if it is a pair) is searched for the first occurrence of an element such that

(EQ **prop** (CAR **element**))

is true. When found, the value of GET becomes (CDR **element**). If such an element

is not found, the value of GET is NIL.

Also see PROPLIST, MAKEPROP, REMPROP, REMALLPROPS.

(INTERN **string**)

This function is used to augment the object list (actually a hash table, but the name object list has historical roots). The character string **string** is sought in the oblist, and if found, the identifier having the given string is returned as the value of INTERN. If a string equal to **string** cannot be found in the oblist, a new identifier structure is created with a print name equal to **string**, and the newly created identifier is returned as the value of INTERN. If **string** is not a character string, an error break occurs.

(MAKEPROP **id propname propvalue**)

Function to update the property list of the identifier **id**. If the **propname** property already exists, its associated value is changed to **propvalue**. If the **propname** property does not currently exist, a new property with this name is put at the beginning of the property list.

The value of **id** must be an identifier, but **propname** and **propvalue** may be any expressions. See also REMPROP, PROPLIST, REMALLPROPS and GET.

(MAPOBLIST **funct**)

This function uses (OBARRAY) to access OBLIST, the LISP data structure which remembers all INTERN'ed identifiers, then applies **funct** to all of the identifiers in OBLIST. The value of MAPOBLIST is NIL.

(OBARRAY)

Returns as value a copy of the current LISP object array. This is a reference vector containing elements which are either identifiers (INTERN'ed variables) or small integers. A small integer zero indicates an available cell in this hash table, a small integer minus one indicates a deleted identifier (result of a use of the REMOB function).

Because the value of OBARRAY is a copy of the actual object array, it may be modified in any way by the user.

(PNAME **id**)

Returns a copy of the print name of **id**. If the value of **id** is not an identifier, an error break is taken. The print name is a character string.

(PROPLIST **ident**)

The value of **ident** must be an identifier. Returns the property list associated with that identifier. The property list is a standard LISP association list. The final CDR of the property list is reserved for LISP/370 system use. See also GET, MAKEPROP, REMPROP, REMALLPROPS.

(REMALLPROPS **ident**)

The value of the argument **ident** must be a normal identifier, not a GENSYM. All properties on the property list of this identifier are removed. The value of REMALL-PROPS is **ident**. See also REMPROP, GET, MAKEPROP, PROPLIST.

(REMPROP **ident name**)

The **name** property of the identifier **ident** is removed from the property list of **ident**. The value of REMPROP is NIL if there is no **name** property. If the property exists, the value of REMPROP is the value associated with that property. See also MAKE-PROP, REMALLPROPS, GET, PROPLIST.

(SASSOC **item alist fn**)

This function is similar to ASSQ, but requires three arguments and if **item** is not matched, it returns the result of applying **fn** to no arguments, instead of the NIL value returned by ASSOC.

(UASSOC **item alist**)

This function is identical to ASSOC except that UEQUAL, rather than EQUAL, is used for the comparison of **item** with the **alist**.

OTHER FUNCTIONS

(AND e_1 ... e_n)

This is a macro which expands into a COND expression implementing the logical and of e_1, e_2, ..., e_n. For example,

(AND e_1 e_2 e_3) = (COND (e_1 (AND e_2 e_3)))

(CHARP x)

This is a macro which expands into code to test whether x is a character object: one of 256 identifiers defined at system generation time which span the total range of possible single character print names. The value of CHARP is NIL if x is not one of these objects, otherwise the value is x.

When interpreted, (CHARP x) is implemented by (,CHARP x) where ,CHARP is the binary program which was obtained by compiling the macro expansion.

(COPY x)

This function copies the structure x, including all of its substructure, and returns the copied structure as its value. COPY will correctly copy any looped, circular structure. This is the function to use when a top–level copy (such as is made by APPEND) is inadequate, or when circular structure is involved.

(CYCLES x)

This function examines an arbitrary object, x, for circular structure. If no cycles are found in x, the value of CYCLES is NIL. If any cycles are found, the value of CYCLES is a reference vector containing two elements for each cycle. The 0, 2, 4, ... elements of the vector are pointers to the list node or vector of a cycle which is closest to the root of the structure, x. The 1, 3, 5, ... elements of the vector are zero and are provided for the caller to use for his own purposes.

See also the description of SHAREDITEMS, which will discover shared substructure not in a cycle as well as cycles.

(CYCLESP x)

This function examines x for circular structure, and returns NIL if there is none, or T if there is some circular structure.

(EQ x y)

EQ tests for pointer identity between its two arguments. Its value is the identifier *T* if x and y are identical pointers. This means that the pointer type codes as well as the pointer address fields are identical. If these fields are not identical, the value of EQ is NIL.

EQ may be used for quick tests of equivalence. If two expressions are EQ, then they are necessarily EQUAL; however, the converse is not true. Two expressions which are not EQ may nevertheless be EQUAL.

Note that two copies of a given object will not be EQ because they are stored at different locations. Similarly, it is possible to have different representations of the same numeric value which are EQUAL, but which are not EQ.

(EQUAL x y)

This is a generalized equality testing function applicable to any LISP objects, including circular structures and numeric quantities.

For numeric quantities to be EQUAL, they must represent the same value. For tests involving one or two real (floating point) numbers, a fuzz factor may be relevant. This is explained in the section of this manual discussing data types. If an integer is to be compared with a real number, the integer is converted to a real value for the comparison.

Two vectors are EQUAL if they are of the same type, the same length, and their absolute parts are identical and their pointer parts are EQUAL.

For composite arguments, EQUAL implements access-equivalent equality testing. This means that two structures are EQUAL if every part of one structure which can be reached by a composition of accessing functions is EQUAL to the corresponding part of the the other structure reached through the same composition of accessing functions. Intuitively, two structures are EQUAL if they denote the same (possibly infinite) tree.

The value of EQUAL is either NIL or *T*.

See the discussion under LISTS in the Data Type section for an example and further commentary.

(FUNARGP x)

If x is a funarg data object, returns x as its value; otherwise returns NIL.

(GENLABEL)

The value of this function is a non-stored constant suitable for use as a statement label inside of a PROG expression or in a LAP contour. These constants are generated in a series which is reset to its starting value by the COMP370 function so that the same values may be reused. The principal use of GENLABEL is by macro definitions which require a locally unique label to be incorporated into their expansion.

GENLABEL's share the peculiar nature of GENSYM's with respect to READ. They are never read in verbatim, but rather each distinct GENLABEL in an expression being read is replaced in the new structure READ produces by a newly generated GENLABEL.

Also see GENSYM.

(GENSYM)

This function constructs a new, unique identifier. This identifier is returned as the value of the GENSYM function, and may be used in any of the contexts suitable for an identifier. In particular, it may be used as a variable (either fluid or lexical).

These GENSYM identifiers are treated specially by the standard PRINT and READ routines, in that they are identifiable as GENSYM's in printed output and when one is read, it is replaced by a new GENSYM in the structure created by the READ program. Thus, if the same expression is read several times, it will contain unique GENSYM's in each copy read, although there will be only one new GENSYM created for each distinct GENSYM in the expression being read. If the same GENSYM occurs more than one time in the input expression, the same newly created replacement GENSYM will be referenced every place the original GENSYM was referenced.

The mechanism used to insure unique ID's is simply to have a counter which is incremented every time a new GENSYM is required and to incorporate this counter's value into the print name of the identifier. A super garbage collector will compact all currently active GENSYM's are reset this counter to the next available number.

There are several factors governing the number of possible GENSYM's possible using this approach, depending upon the exact implementation. Since this will probably change once a super garbage collector is available, it seems prudent to state at this time only that there is a maximum of no less than 786,433 of these GENSYM's before any implementation limit is reached.

Because there are a limited number of GENSYM's, it is recommended that GENLABEL's be used whenever they are appropriate rather than the more costly GENSYM's.

(MSUBRP x)

Returns x it x is a compiled macro (MLAMBDA ...) expression, otherwise returns NIL.

(OR e_1 ... e_n)

This is a macro which generates a COND expression implementing the logical "or" of e_1, e_2, ..., e_n. For example,

(OR e_1 e_2 e_3) = (COND (e_1) (e_2) (e_3))

(NONSTOREDP x)

Returns x if x is not a stored object. This means, in a somewhat arbitrary way, that the type code of x has a zero in its high-order bit. Thus, small integers, generated symbols, binary programs are examples of non-stored objects.

(PLACEP x)

Returns x as its value if x is a read-place-holder, else returns NIL. The usual application of this function is to detect when READ has encountered an end of stream condition, since this is the only condition under which READ will return a read-place-holder as its value. In other circumstances, READ uses these objects internally to mark where in the structure it is building a substitution must be made to establish the required sharing. Thus, in these normal cases, all read-place-holders have been replaced with their proper values.

(RESETQ *id* **value**)

This macro expands into code which returns the current value of the variable *id* while assigning to it the value of **value**. The *id* argument is not evaluated, so use of this macro resembles SETQ except for the value of the expression.

This macro is typically invoked when the user wishes to pass some large data structure to a subfunction which will operate on it, and the calling function has no further need for the original value. Using RESETQ may achieve an economy in the maximum storage required at one time for LISP data. For example,

(RETURN (**foo** (RESETQ **a** ())))

invokes the function **foo** with the current value of **a**, but assigns NIL to **a** before passing control to **foo**. Thus, if **foo** during the course of its execution were to assign some other value to the variable used to bind its argument value, there might be no

outstanding references to the original value of **a**, and a garbage collection would recover the storage which would otherwise be retained by the variable **a** in the calling function.

(SHAREDITEMS x)

This function examines the arbitrary structure **x** for shared substructure. If there is only one possible path from the root, **x**, to every part of **x**, then the value NIL is returned. If there is some shared substructure, the value of SHAREDITEMS is a reference vector whose 0, 2, 4, ... elements are pointers to the shared nodes (either vectors or list cells), and whose 1, 3, 5, ... elements are small integers indicating the minimum nesting depth of the corresponding node from the root of the structure.

Since any cycle must be shared, this function subsumes CYCLES, which is useful when only circular structure is of interest.

(SUBRP x)

Returns **x** if **x** is a compiled function, otherwise returns NIL.

(TEMPUS–FUGIT)

This function returns as value the amount of CPU time, in milliseconds, used by the user in the current logon. Its value is a small integer.

(TYPEBYTE x)

This function returns as a small integer the eight bit type code of **x**.

(UEQUAL x y)

This is a generalized update-equality testing function applicable to any LISP object in the same sense as EQUAL. It differs from EQUAL in that for two structures to be UEQUAL, not only must corresponding parts of the structures be EQUAL through the access functions, but there must be the same number of unique parts and if any of these parts were to be updated in one structure and the same update operation performed on the corresponding part of the other, then the structures would still be EQUAL.

In addition, numeric values are considered UEQUAL only if they are of the same type and numerically equivalent. Bit and character strings are UEQUAL only if they have the same capacity as well as the same type, length, and contents.

Intuitively, two structures are UEQUAL if and only if they denote equivalent rooted directed graphs, i.e. if they denote EQUAL structures which also have the same acyclical and cyclical sharing structure.

The value of UEQUAL is either NIL or *T*.

See the discussion under LISTS in the Data Type section for an example and further commentary.

DEBUGGING FACILITIES

(? "**command** ["A] ["NPP] [**stream**] [**smint**$_1$ [**smint**$_2$ | *]] [**sd**$_1$ [**sd**$_2$]])

EXAMINE STACK FRAMES

The ? function controls the examination of stack frames created by the execution process. It is possible to look at arguments of functions, to determine bindings of variables, and to display variable values.

Only the first argument is required; the order of the remainder is not significant.

ARG1: **command**: an identifier specifying the command, i.e., the kind of stack examination desired. The possibilities are:

IDENTIFIER	RESULT
INDEX	Series of stack frame identifications sequentially indexed with a small integer. These appear in a LIFO order (last in execution, first in listing). The form of each frame identification is: 1 index, or frame number 2 frame name, or NIL 3 frame type 4 contour level, if not outermost
FULL	Stack frame identification followed by 1 argument names, if any arguments exist 2 all variables, with their values, which have been bound at this frame.
BIND	Stack frame identification for only those frames at which some binding information exists; following each identification are all the variables bound at that frame.
ARGS	Stack frame identification for only those frames representing functions to which arguments have been passed; following each identification are all the argument names and their values.

	LEX	Stack frame identification for only those frames at which lexical binding information exists; following each identification are all the variables, and their values, bound lexically at that frame.
	FLUID	Stack frame identification for only those frames at which fluid binding information exists; following each identification are all the variables, and their values, with FLUID bindings at that frame.
	SECD	Stack frame identification for only those frames associated with SECD (interpretive) execution and which also have some elements on the SECD control or stack; there follows the elements of the control and stack.
	(list of identifiers)	For every identifier there is an indication of lexical or fluid binding, stack frame identification, and value. If the identifier refers to a generated symbol, i.e. one in the form %Gn, only the numeric part, i.e. n, should appear in the list.
ARG2:	"A	Indicates that the Access chain of frames is to be examined. If not present, the control chain is examined.
ARG3:	"NPP	Indicates that No Pretty Printing is to be done. This option is useful when condensed output is desirable. It is essential when examining stack frames after detection of an "insufficient heap" condition.
ARG4:	**stream**	Specifies the stream to which output from the ? function is to be directed. For example, by use of an appropriately defined stream, output could be directed to a disk file. When convenient, this file can be printed off-line or otherwise examined.
ARG5:	**smint**$_1$	Specifies the first stack frame which is to be examined.

ARG6:	**smint**$_2$	Specifies the last stack frame which is to examined.
	*	Specifies that the stack examination should continue through to the highest level.
ARG7:	**sd**$_1$	Specifies that the stack examination should be relative to the stack frame referred to by this State Descriptor.
ARG8:	**sd**$_2$	Specifies that the stack examination should terminate with this frame; this is true even if the value set by the integer specifying the last frame is deeper into the stack. Actually, the stop condition is more complex than this. The display terminates whenever a frame is encountered which is also part of the control or access chain (according to the INDEX argument of ?) emanating from **sd**$_2$. The termination test is performed in the following way. Each time a new stack frame is to be displayed, a search is made starting at the stack frame referenced by **sd**$_2$. The search follows either the control or access chain back from this starting frame, comparing the head-of-environment in the new frame to be displayed with the head-of-environment of the current frame in the **sd**$_2$ chain. If the heads-of-environment are identical, the stop criterion is met and the stack display function is finished. If the heads-of-environment are not identical, the search continues with the next frame up the chain designated by **sd**$_2$ until either the stop condition is met, or the end of stack is reached. In the latter case, the new frame is displayed, the access or control chain from that frame is ascended, and the process iterates.

(MONITOR id [entry-list [exit-list]])

MONITOR is a simple call tracing function. The first argument is the **id**, the value of which is the function or macro to be traced. Once MONITOR has been executed, all calls to **id** will be intercepted and the values of the arguments, followed by the value of the call to **id**, or the expansion of **id** if it is a macro, will be printed on CUROUTSTREAM. If **id** is MONITORed in compiled code, the **id** of the calling program will

also be printed.

The two optional arguments are lists of *id*s. If they are present and non-NIL the *fluid* bindings of the *id*s in the **entry-list** will be printed before the MONITORed function is actually called, while those in the **exit-list** will be printed after the function returns.

The affect of MONITOR is removed by (UNEMBED **id**).

NOTE: EQUAL and UEQUAL may not be MONITORed; a system error will result if an attempt is made to do so.

DEFINE, COMPILE, and ASSEMBLE

The two basic "operators" (actually basic macros) used for function and macro representation are, respectively LAMBDA and MLAMBDA. A number of auxiliary functions and a macro exist in the LISP370 system to aid in function definition. In addition, there exists a fluid variable, OPTIONLIST, the value (and binding) of which both affects and is affected by the definition functions.

The relevant functions are:

DEFINE
TEMPDEFINE
COMP370
LAP370

The macro is:

LAM

LAMBDA and MLAMBDA

Both LAMBDA and MLAMBDA expressions have the form

([M]LAMBDA *bv body*)

where *body* represents any LISP expression and *bv* represents a bound variable structure.

The form and meaning of the *bv* part of the LAMBDA expression, simply called *bv* in what follows, has been extended in LISP370. (The following discussion applies directly to LAMBDAs. With slight modification, to be set forth below, it also applies to MLAMBDAs.)

We will define "variable designator" to mean either an identifier or one of the special forms

(FLUID *id*)
(LEX *id*)

The *bv* of a LAMBDA expression is a variable designator, a constant or any non cyclic list structure of variable designators and constants.

When a LAMBDA expression (or its compiled surrogate, a binary program image, or *bpi*) is APPLXed to a list of values (arguments), the list or its elements are placed in correspondence with the *bv* or its elements. This can best be explicated by a set of examples, all of which will be described with respect to the argument list:

(QUOTE ((A B) C ((D E . F) G)))

bv = NIL

variable	value
none	

No binding takes place. No variable is named in the *bv*, and thus no binding is made and the argument list is discarded. This is not dependent on the *bv* being NIL, any constant value for the *bv*, e. g. 12, would result in the same behavior.

bv = X

variable	value
X	((A B) C ((D E . F) G))

A *bv* consisting of a single variable designator causes the entire argument list to be bound to the designated variable.

bv = (FLUID VAR)

variable	value
VAR	((A B) C ((D E . F) G))

If the variable designator is of this form the resulting binding is fluid. That is, free occurrences of this variable farther down the access chain evaluate to the binding mentioned here, providing no further FLUID bindings of this variable intervene.

bv = (X)

variable	value
X	(A B)

Remembering that (X) is a shorthand for (X . NIL) we have the CAR of the argument list matched with and bound to X, while the CDR of the argument list is matched with NIL, a constant, and therefore discarded.

bv = (X . Y)

variable	value
X	(A B)
Y	(C ((D E . F) G))

In this case the CDR of the argument list is matched with, and thus bound to Y, the CDR of the *bv*.

bv = (U V W)

variable	value
U	(A B)
V	C
W	((D E . F) G)

This is the most usual case (and the only case in "traditional" LISP). The *bv* is a list of identifiers, with one identifier for each item in the argument list.

bv = ((FLUID THIS) (LEX FLUID) THAT)

variable	value
THIS	(A B)
FLUID	C
THAT	((D E . F) G)

Here the binding of THIS is accessible from below. The need for the form (LEX id) is shown, as only thus can a variable named FLUID be bound (see the next example).

bv = (THIS FLUID THAT)

variable	value
THIS	(A B)
THAT	(C ((D E . F) G))

The catch here is that (THIS FLUID THAT) is a short hand for (THIS . (FLUID THAT)). Thus, given the definition of variable designators, (FLUID THAT) is a variable designator which matches the CDR of the argument list, rather than the second and third variable names.

bv = ((U . V) W ((NIL NIL . X) . Y))

variable	value
U	A
V	(B)
W	C
X	F
Y	(G)

Here we have the general case of argument decomposition. Note that items in the argument list corresponding to NILs (constants) in the *bv* are simply dropped.

bv = (W (X . Y) Z)

variable value
not conformal

No binding takes place. This case results in an error, as the form (X . Y) in the *bv* is matched to an atom in the argument list, and thus the required decomposition cannot be made.

bv = (W X Y Z)

variable value
not conformal

No binding takes place. This also results in an error, in theory the same as in the last case (the variable designator Z in the *bv* is being matched with the CADDDR of the argument list, which does not exist as the CDDDR is NIL), but in practice the two errors are distinguished in compiled code, this being the "non-conformal application" error, the previous being "illegal CAR" error. The interpreter, which is not bothered by the problem of generating code at one time for execution later, treats both errors as "non-conformal applications".

bv = (X Y Z 1)

variable value
not conformal

No binding takes place. This results in the identical error as the previous example. Even though the fourth item in the *bv* is a constant and thus does not result in a binding, the *bv* and the argument list must still be conformable.

All of the above applies to MLAMBDAs with the understanding that the argument list is the original form whose CAR gave rise to the MLAMBDA (or *mbpi*). Thus to obtain this form itself (as in the old LISP system) the expression

(MLAMBDA [*id* | (FLUID *id*) | (LEX *id*)] *body*)

must be used.

LAM

Nominally SETQ should be sufficient for function definition. Since the interpreter applies the value of the operator, rather than using EXPR, FEXPR, etc. properties, the assignment of a LAMBDA expression to a variable serves to establish that variable as a function. It would seem, then, that DEFINE is not needed in this system. Efficiency (by a rather devious argument) makes such a function desirable.

One feature of most existing LISP systems which is not fundamental in LISP370 is the function which receives its arguments unevaluated. These are usually known as FEXPRs or FLAMBDAs. Such functions can be realized via LISP macros; however, such macros are both stereotyped and somewhat complex, a combination of properties which lends itself to automatic generation. To achieve this, a single macro, LAM, is provided.

Expressions can be written of the form (LAM *bv* body*), where *bv** represents an augmented bound variable list, containing various types of declarative information. The LAM macro expands to a MLAMBDA expression (not in all cases, the exceptions will be dealt with shortly) which processes the variables according to the declarations in the *bv* list and then applies a LAMBDA expression incorporating the *body* from the original LAM.

For example:

((LAM ((QUOTE X) Y) (LIST X Y)) A B)

would first expand to:

((MLAMBDA L *e*) A B)

(where *e* is a tailored expression based on the exact form of the bound variable list and the *body* of the LAM), which in turn would become:

((LAMBDA (X Y) (LIST X Y)) (QUOTE A) B)

The form:

(LAM ((QUOTE U) (QUOTE V) (QUOTE W)) *body*)

is identical in affect to the old form:

(FLAMBDA (U V W) *body*)

The catch in this story is that the form (LAM *bv body*) is not recognized by the interpreter as a unusual object. Thus, if an identifier, say Q, were to be given (LAM *bv body*) as its value the evaluation of (Q arg1 arg2 ...) would result in a "dynamic macros not allowed" error. If the value of Q were the *value* of the LAM expression this would work, as the value would be a FUNARG with the appropriate MLAMBDA expression for its expression part.

The creation of a FUNARG, however, involves saving a state, which in turn impacts (albeit slightly) the performance of the system. Furthermore, if the resulting MLAMBDA were to be compiled it would contain the entire *body* of the original LAM as a QUOTEd s-expression. This would not only consume heap space, but would, upon compilation of instances of Q, replicate the code resulting from the evaluation of the *body*.

The alternative to these conditions is to write a DEFINE function to "mess" with the MLAMBDA generated by the LAM. The LAM is expanded (by use of the system operator MDEFX) and the QUOTEd LAMBDA expression is extracted from it. A system generated

name is generated and inserted in place of the LAMBDA expression and the system generated name is then given the LAMBDA expression as its value.

Taking the example given above;

(LAM ((QUOTE X) Y) (LIST X Y))

is the value of Q. The result will be to define Q as

(MLAMBDA L *e*)

as before, but the *body* of the MLAMBDA, *e*, will differ from the previous case. In addition a new identifier, U,Q,23 say, will be created and given a value of

(LAMBDA (X Y) (LIST X Y))

The expansion of an instance of (Q A B) will now become, due to the modified MLAMBDA,

(U,Q,23 (QUOTE A) B)

Both DEFINE and the compiler know about the peculiarities of LAM and both perform this transformation. Note that LAM may be re-defined by the user, but only at his peril. There are various interactions between LAM, DEFINE, and various parts of the compiler which must all be considered if any change is to be made.

As was stated above LAM does not always expand to a MLAMBDA. In such cases the transformation described is not possible, and the expansion (which ultimately results in a LAMBDA expression) is assigned to the name (in the case of DEFINE) or compiled.

(LAM *bv body*)

The LAM macro is used in defining functions having declarative information in their bound variable lists. At the moment three types of such information is processed, QUOTEd arguments, restricted variables and equated variables.

Although the *bv* part of LAMBDA expressions has been generalized from a simple list to an arbitrary tree structure, there is still (as its name implies) a basic "listness" about the argument list. For each item in the argument list, or for the CD*R of the list there must correspond an element, either a variable designator or a structure, in the *bv*. In order that an argument be passed unevaluated to the binding mechanism the structure in the *bv* corresponding to that argument must be "wrapped" in a QUOTE (see the following examples). Thus:

(LAM (QUOTE X) *e*)	indefinite number of arguments, all QUOTEd
(LAM (X (QUOTE Y)) *e*)	two arguments, the first evaluated, the second QUOTEd

(LAM ((QUOTE (X . Y))) *e*)	one unevaluated argument which is to be decomposed, with its CAR bound to X and its CDR to Y
(LAM (X . (QUOTE Y)) *e*)	one evaluated argument, indefinite number of trailing QUOTEd arguments

Note that the last case (as the trailing FLUID binding discussed under LAMBDA) would print as

(LAM (X QUOTE Y) *e*)

a fact which sometimes causes consternation in the ranks.

The second type of declarative information processed by the LAM macro is restrictions on variables. (In fact, restrictions and equatings are processed by two other macros, LAM,1 and LAM,2; however, they can be safely ignored and only the top level LAM used in all cases.) In order to introduce restrictions (and the soon-to-be-described equated variables) we introduce two new forms which behave like variable descriptors in the basic *bv*.

A restricted variable is declared by writing the form

(: var-des *e*)

where var-des is a variable designator, and *e* is any LISP expression. This form is matched to a part of the argument list (as if it were a variable designator in its own right) and the binding takes place as if the contained variable descriptor had stood in the place of the form. After the binding the expression is evaluated and if its value is NIL an ERROR call is made. Note that the *e* can (and almost always will) contain references to the variable being bound. Further, as it may be any LISP expression, it can change the value of the binding.

Other variables bound in the *bv* should, in general, not be referenced in the expression, as the order of binding and testing of restrictions is highly dependent on the exact structure of the *bv*.

For example

(LAM ((: X (PAIRP X)) (: Y (NUMBERP Y))) *e*)

will bind two arguments to X and Y, test the first for pair-ness and the second for number-ness, and signal an error if either fails its test.

(LAM ((: X (COND ((ATOM X) (SETQ X (LIST X))) (T T)))) *e*)

binds one argument to X and, if X is bound to an atom, CONSs X to NIL. The restriction is always satisfied, and, during the evaluation of *e*, X is always bound to a pair, whatever the original argument's value.

The final declarative form processed by LAM is an equated variable.

An equated variable allows a part of the argument list to be bound to more than one variable, or for a part of the argument list to be both bound to a variable and to be decomposed and bound to a structure of variables. The form for an equated variable is

(= {var-des | restriction} *bv*)

As in the restricted variable, this form acts as a variable designator in the matching process. The corresponding part of the argument list is bound to the variable indicated by the var-des or restriction element of the equating form. If a restriction is present, it is tested and (if successful) the part of the argument list which has been bound to the variable is then bound to the *bv* part of the equating form, as if they were the argument list and *bv* of a LAM (save that (QUOTE structure) forms are not allowed).

(LAM (= X (Y Z)) *e*)

Causes the entire argument list to be bound to X while the first two arguments are bound to Y and Z respectively. Note that an error will occur if the argument list has fewer than two members.

(LAM ((= (: X (PAIRP X)) (A . B))) *e*)

binds a single argument to X and tests it for pair-ness. If it is, not a pair an error is signalled; if it is, its CAR is bound to A and its CDR to B. This is a case where the restriction demonstrated above, which forces pair-ness on its variable, could be used. Thus

```
(LAM ( (=
            (: X (COND ((ATOM X) (SETQ X (LIST X))) (T T)))
            (A . B) ) )
                e)
```

OPTIONLIST

The values of various properties on this augmented list control various aspects of definition, compilation and assembly. GET is used to search OPTIONLIST, thus NIL is the default value for any property not explicitly present. Currently meaningful properties are:

EXPANSIONSTATE

The state in which macros are to be expanded by the compiler. The use of EXPANSIONSTATE protects from conflict between variables bound by the compiler and the bindings of macros used in expressions being compiled. If the value is NIL, the initial state (where only nil-environment bindings are present) is used.

NOLINK

If the NOLINK property has a non-NIL value the result of an assembly is not made into a *bpi*. The default is to creat a *bpi* and assign it to the *name* with which it was paired in the specification list.

INITSYMTAB

The value of this property should be an association list (or NIL, which is an empty association list) which will be searched by the assembler (LAP) for operation code values, symbolic register names, symbolic immediate operand names and symbolic literals. The values in INITSYMTAB override the build-in values of the assembler, and are overridden in turn by symbols established by EQU statements in the LAP code.

NONINTERRUPTIBLE

If the NONINTERRUPTIBLE property is non-NIL, no polling for interrupts is inserted in the *bpi* by the assembler. Explicit POLL statements will be assembled. The default value is NIL, i. e. the *bpi* is interruptible.

SOURCELIST

If SOURCELIST has a non-NIL value the source program (either LISP or LAP) is PRETTYPRINTed by the definition functions. The default is NIL. When running with SOURCELIST non-NIL it should be remembered that the printing by the supervisor can be controlled by the settings of the *fluid* variables ,ECHOSW and/or ,VALUSW.

TRANSLIST

A non-NIL value for TRANSLIST causes the output of pass one of the compiler to be PRETTYPRINTed. This is the "transformed" LISP, with all macros expanded and with various other changes, which will be made into a *bpi* by pass two of the compiler and the assembler. A number of forms internal to the compiler appear in this listing. If definitions of these internal forms existed (unfortunately impossible in some cases) interpretation of this transformed LISP would duplicate the behavior of the *bpi* which results from the full compilation/assembly process. The default value is NIL.

LAPLIST

A non-NIL value for LAPLIST causes the assembly code produced by the compiler to be PRETTYPRINTed. This property does not control the printing of LAP source code, which is under the control of the SOURCELIST property. The default value is NIL.

BPILIST

A non-NIL value for BPILIST causes an assembly listing to be produced. This listing contains the hexadecimal System/370 machine code produced by the assembler, together with a variable amount of symbolic LAP code.

If BPILIST is non-numeric or if it is greater than 3 a full listing is produced. This includes all instructions generated by the assembler or by LAPMACROs, all comments and all source instructions.

A value of 3 causes intermediate instructions to be dropped from the listing. That is, instructions generated by the assembler or LAPMACROs which in turn resulted in the generation of further instructions rather than in object code.

A value of 2 causes comments to be dropped from the listing.

A value of 1 causes only source instructions to be printed symbolicly, although the object code (hexadecimal) is printed in full.

LISTING

The value of LISTING should be NIL or a fast stream. If LISTING is a stream the output produced as a result of the preceding four options (SOURCELIST, TRANSLIST, LAPLIST and BPILIST) will be written onto that stream. If LISTING is NIL it defaults to the value of the *fluid* CUROUTSTREAM.

MESSAGE

The value of MESSAGE should be NIL or a fast stream. All error and warning messages from the definition functions are written onto the MESSAGE stream. If the MESSAGE and LISTING streams are not EQ the MESSAGE steam is made to dominate the LISTING stream. (See the DOMINATESTREAM function.) If the value of MESSAGE is NIL it defaults to CUROUTSTREAM.

FILE

The value of FILE should be NIL or a stream. If FILE is non-NIL a loadable *bpi*-image will be written onto it. If FILE is NIL no action is taken.

By use of the FILE and NOLINK options programs can be compiled and/or assembled for future loading without their being defined in the running system. (See LOADFILE function.) The resulting file, when loaded, causes the same assignments and/or MAKEPROPs to take place as would have resulted if the NOLINK option were NIL. The current form of a loadable file may be changed in the future, however this is still a matter of dispute and discussion.

TYPE

The TYPE option is set internally by the definition functions and is used to control the final disposition of the resulting *bpi*, whether it is to be a function, a LAPMACRO, etc.

DEFINITION FUNCTIONS

The definition functions all need the same general arguments, a required list of function specifications and an optional local parameter list.

The function specification list in turn has two forms (solely for convenience), either a list of length two consisting of a identifier (*name*) and an expression (*form*), or a list of such lists. The latter is analogous to the argument to COMP360, MACRO and DEFINE in the LISP/360 system. The allowed value for the *form* portion of the specification and the processing it undergoes is dependent on which of the definition functions is being called.

The local parameter list, which may be omitted, is used to augment and override the value of the current binding of OPTIONLIST. In the definition functions OPTIONLIST is bound, as a *fluid*, to the local option list, APPENDed to the front of the existing *fluid* binding of OPTIONLIST. Thus, when a parameter value is required from OPTIONLIST, the local values take precedence over previously present values.

(DEFINE **specification-list [option-list]**)

Uses free:
OPTIONLIST

Binds fluid:
OPTIONLIST OUTERSTATE EXPANSIONSTATE LISTSTREAM MESSAGESTREAM SOURCELIST TRANSLIST LAPLIST BPILIST

DEFINE tests the *form* parts of its specification list for (LAM *bv body*), and if any are found they are "split" as discussed above. The resulting forms (or the original forms if not LAMs) are then assigned as values to the corresponding *name* parts of **specification-list**. This assignment takes place in the caller's environment. The *forms* can be any s-expression.

(TEMPDEFINE **specification-list [option-list]**)

Uses free:
OPTIONLIST

Binds fluid:
OPTIONLIST EXPANSIONSTATE LISTSTREAM MESSAGESTREAM SOURCELIST TRANSLIST LAPLIST BPILIST

TEMPDEFINE adds a new EXPANSIONSTATE property to the current *fluid* binding of OPTIONLIST. It is added by CONSing rather than by MAKEPROP, thus not disturbing any previous EXPANSIONSTATE property. This new state is the previous EXPANSIONSTATE property augmented by bindings of each of the *names* in **specification-list** to the corresponding *forms*. The effect is to make these definitions

available during the macro expansion process of any compilations done while this EXPANSIONSTATE property is accessible. (See the discussion of macro expansion under COMP370 as well as the explanation of the use of OPTIONLIST.)

(COMP370 **specification-list [option-list]**)

COMP370 is the function used to produce *bpi*s and *mbpi*s from LISP expressions. The exact processing to be done on **specification-list** is dependent both on the *form*s in the list and on the values of various properties on the local option list (if any) and on the current *fluid* binding of OPTIONLIST.

First the EXPANSIONSTATE is augmented by bindings of each of the *form*s to its corresponding *name*. This is to allow the functions and macros being compiled to be available during macro expansion. Then each *name/form* pair is processed in turn. If the *form* is a LAMBDA or MLAMBDA expression the compilation proceeds.

If the *form* is an LLAMBDA, i.e.,

(LLAMBDA *bv body*)

the form

(LAMBDA *bv body*)

is constructed and compiled, and the resulting value is a form

(LLAMBDA *bpi*).

Otherwise the CAR of the *form* is examined. If it is an identifier bound to a MLAMBDA-expression or a *mbpi* in the EXPANSIONSTATE or if it is an explicit MLAMBDA-expression the *form* is macro expanded and the processing is repeated. If it is not a LAMBDA or MLAMBDA expression and fails to macro expand to one it is treated as an expression.

To compile an expression *form* is replaced by

(LAMBDA () *form*)

which is then compiled. the resulting *bpi* is not made the value of the corresponding name, unlike the case for a compiled LAMBDA or MLAMBDA expression. Rather, the *name* is given as its value the single element list

(LIST *bpi*),

where *bpi* is the result of the compilation process on the constructed LAMBDA expression. The *bpi* is compiled "open", that is, it has access to lexically bound variables in the environment from which it is invoked, in contrast to normal LAMBDA

expressions which have access only to *fluid* bound variables. When the *name* is applied the interpreter is entered and the value of the *name* is re-evaluated, resulting in the value which the original expression would have yielded.

In all cases the compilation proceeds though both passes and the result is assembled. If errors occur during either the compilation or assembly neither *bpi* nor *bpi*-image is created, regardless of the values in OPTIONLIST.

(LAP370 **specification-list [option-list]**)

Uses free:
OPTIONLIST

Binds fluid:
OPTIONLIST EXPANSIONSTATE LISTSTREAM MESSAGESTREAM SOURCELIST TRANSLIST LAPLIST BPILIST S,ERRORLOOP ,LABELNUM

LAP370 assembles and may, depending on the values in OPTIONLIST, create *bpi*s and/or loadable *bpi*-images of the assembly language (LAP) programs in its specification list. See the section on LAP370 for a description of valid LAP programs.

The LISP Compiler

*CODE and FR*CODE Expressions

These are expressions designed to allow the inclusion of LISP Assembly Program statements in a LISP program which is to be compiled. They are extensively used in the system implementation, but because there is no way for the compiler to check on the reasonableness of the LAP instructions given, it is very easy to violate some convention of the LISP execution environment, leading to failure of the LISP system.

The reason for using such dangerous constructions at all is based on the argument for efficiency. Many of the primitive LISP functions are implemented by such expressions.

(*CODE *e free-list LAP-stmt** ...)

When interpreted, the value of this form is the value of the expression *e*. The remainder of the form is ignored.

When compiled, the (expression *e* is ignored and the value of the form is the contents of register *S1 following execution of the LAP statement(s). The *free-list* may have any of three formats:

NIL - indicates that the LAP code is "safe" (i.e. no state-saving may occur).

(*T*)- indicates that the LAP code is "unsafe"; state saving may occur. This is the typical value if a normal LISP function call is made, and forces the surrounding contour to be raised.

(*s-flag id* ...) - The LAP code is "safe" or "unsafe" if *s-flag* is NIL or non-NIL, respectively. The identifier(s) following s-flag are referenced "free" by the LAP code (i.e. they are not bound by the innermost surrounding contour).

*CODE expressions are just that: expressions. Thus a form such as:

((*CODE ...) e_1 e_2 ...)

causes the value of the *CODE expression to be APPLYed to the arguments e_1, e_2, ...

((FR*CODE e_0 *f-list LAP-stmt* ...) e_1 ... e_n)

The effect is the same as interpreting

(e_0 e_1 ... e_n)

where the limitations on the nature of e_0 arise from the fact that all of the arguments e_1, ... e_n are evaluated before e_0 is applied.

When compiled, the arguments are evaluated and all but the last are pushed onto the stack. The last argument is left in register *S1. Following the evaluation of the arguments, the LAP statement(s) are executed. Finally, a POP of n-1 is assembled (automatically). Thus the LAP code should reference the PUSHed arguments (if any -- the most common use of this construction is when there is but one argument, in which case it is not PUSHed onto the stack) by the (TOP **n**) notation (see the section describing LAP).

The *f-list* element has the same use as in a *CODE expression. Note that an FR*CODE expression is valid *only* in operator position.

LAP -- The LISP Assembly Program

LAP/370 (to be called simply LAP from here on out) is a mid-level assembly language for programs which are to run the LISP/370 environment on an IBM System/370. LAP produces modules which can be transformed by the LISP/370 Module Loader into *bpis* (binary program images) for use in the LISP system.

LAP programs have the same general structure as LISP programs (see LISP documentation). The basic LAP program is a CONTOUR which corresponds to a LISP (PROG ...) or (LAMBDA ...). It establishes a set of variable bindings, and may combine the functions of the LAMBDA and the PROG by defining both pre-valued variables (arguments) and local, initially NIL-valued variables (PROG variables).

CONTOURS may be nested. When they are, they establish scope for both variables and labels (a label, in this description of LAP, is an identifier used to reference a storage location within a LAP program). A CONTOUR is said to create a contour, which may be either raised or flat. This distinction is dealt with below, in the discussion of the augmented machine. Code within a contour may not reference labels defined in either an enclosing or an enclosed contour. It may reference variables bound by an enclosing contour, provided they have different names than any variables it itself has bound, but it cannot reference variables bound by an enclosed contour. These are the same restrictions as hold for LISP programs, with LAMBDAs and PROGs defining contours.

A LAP program may also include SECTIONs. Their meaning is analogous to the SEQ of LISP: they establish scope for labels, but have no effect on variables. Thus code in a SECTION may reference labels defined in an enclosing SECTION (a CONTOUR subsumes the semantics of a SECTION), but may not reference labels defined in an enclosed SECTI ON. Of course, labels defined in another contour cannot be referenced in any case. A SECTION is said to define a level.

The machine which is the target for LAP code is a S/370 with the LISP stack and environment added, with a predetermined register usage, and with the basic LISP system installed. This basic LISP system includes a number of addressable (EXTERNAL) constants and "foul called" routines (i.e. routines not called by the standard evaluator and a standard calling mechanism). LAP assumes the burden of housekeeping for the environment and the stack.

The world seen by a LAP program upon its being entered is as follows.

REGISTERS

Upon entry certain of the general purpose registers have pre-defined values, while others have undefined values. These values, together with the normal usage of the registers, is outlined in the accompanying table.

The registers *BASE, *HEADE, *TAILE, *DUMP, *FIX, *NIL and *HEAP should be valid at all times.

General Purpose Registers

Register	LAP name	initial contents	normal use
0	*R0	undefined	unrestricted
1	*FRT	offset pointer to last word allocated in stack	indicates the current STACK frontier
2	*SCRATCH	undefined	unrestricted
3	*BASE	address of beginning of *bpi*	addressing for the first 4096 bytes of the program
4	*HEADE	offset pointer to environment head	addressing for arguments and most local variables
5	*TAILE	offset pointer to DUMP of predecessor in the environment chain	access to variables bound by programs up the environment chain
6	*DUMP *STACK	offset pointer to the stack	access to the absolute stack and the pointer stack
7	*NIL	pointer to the NILSEC, the object NIL	access to NILSEC, the object NIL
8	*RET	undefined	linkage register for CALL's, addressing in code beyond the beyond the first 4096 bytes
9	*FIX	address of FIXEDSEC, R/O constants and code	access to the FIXEDSEC routine and constants
10	*SCR2	undefined	unrestricted
11	*S1	undefined	unrestricted internally, value on exit
12	*S2	undefined	unrestricted
13	*S3	undefined	unrestricted
14	*S4	undefined	unrestricted
15	*HEAP	address of the first unused byte in the heap	used for allocating space in the heap

Figure 8

The register *FRT must be valid at function call time, whenever the garbage collector is

called, and whenever an interrupt can be serviced (see TRA, below). The register *FRT is changed by every normal LISP function call. Its value at the time of the call is preserved in the caller's stack frame, but it is not restored by the function exit routine when a return is made to the calling program. The reason for this is simply that, even if *FRT were restored, it is highly likely that the stack frontier would change (due to a PUSH or POP instruction) before the next time the contents of *FRT were required to be valid. Therefore, *FRT is not restored, but rather it is assumed a correct value will be loaded into this register before any operation which may require it to be valid.

The register *RET may only be used freely on the zeroth page of a program, and will be changed by any function call.

The floating point registers (LAP names: *F0, *F2, *F4 and *F6) are freely available within a contour. Their value is undefined upon program entry and may be lost on crossing a contour.

The only registers automatically preserved by the function linking routine across function calls are *BASE, *HEADE, *TAILE and *DUMP. There is a system-wide assumption (which is violated at the risk of complete loss of system integrity) that the values in registers *NIL, *FIX and *HEAP are always correct.

HEAD OF ENVIRONMENT

The head of environment (or *hE*) is a block of 2048 or fewer bytes of storage containing eight bytes (two words) of "housekeeping" information, and the values of all variables bound by the last raised contour. The byte beyond the high (in the machine's address space) end of the *hE* is 2048+4 bytes beyond the address in register *HEADE. The resolution of references to variables in the *hE* is done automatically by the LAP assembler.

Note that a new *hE* and a new stack are created every time a raised contour is entered, and are discarded upon exiting from such a contour. An associated *hE* and stack are called a stack-frame.

STACK

The stack (or DUMP) contains 32 bytes (eight words) of "housekeeping" information (see the DSECTs in the LISP source code for more data). It also contains arguments being prepared for further program calls, indirect pointers to variables bound on enclosing raised contours, or by calling programs, and the saved, previous, values of FLUID variables bound by this contour. If this is a flat contour, the stack contains the values of any variables bound by it. Thus no new *hE* and stack are created when a flat contour is entered.

The stack is also addressed via an offset register, *DUMP (or *STACK), but the register points 2048+4 bytes below the stack's low end, thus allowing the stack to grow upwards. LAP provides constructs for referencing objects on the stack, but only in a static way. This will be expanded upon when the PUSH and POP pseudo-op/arguments are discussed. There is

also a construct called the abstack (absolute stack) which is used to hold non-pointer objects during computations. The abstack is implemented as a section at the base of the pointer stack, however in a LAP program it can be thought of as an independent entity. It is also managed statically, using PUSHA and POPA pseudo-op/arguments.

BPI (Binary Program Image)

The result of a LAP assembly is a *bpi*. This is a LISP binary vector containing the S/370 object program, certain information needed during the call, and a display, which provides a mapping from variable names to locations in the *hE* and the stack.

The header information, which is generated by LAP, comprizes the first 32 (20x) bytes of the *bpi*. The display is located at the end of the *bpi*, and is accessible via a pointer (relative to the origin of the *bpi*) in the header. Neither of these areas need concern anyone programming in LAP. For detailed information see the DSECTs in the LISP system source listing.

Upon entry the register *BASE (3) is used as a base register. LAP programs may be longer than 4096 bytes, however, and the register *RET (8) is used for addressability in those cases. As *RET is also used for function calls some care must be taken in its handling. LAP provides certain services in this regard, see TRA, CALL and FIXRET below, in the section on extended instructions.

At the end of each page (4096 byte section of program), or immediately before the display on the final (or only) page, there is a literal area. These literals are of two types, first simple constants needed in the program, and secondly pointers to communication cells in NILSEC, e. g. QUOTE cells, value cells, and shallow binding cells. The forms which generate these literals are described below. As all pages of a program have addressability to themselves and to the zeroth page, sharing of literals is provided to the greatest extent possible, with literals existing on page zero being shared with all other pages.

LAP STATEMENTS

CONTOUR

Every LAP program has the form

(CONTOUR
declaration
*lap-statement**)

The declaration is a list of items specifying various properties of the LAP code which follows. Its form is

(*function-type*
result-type
argument-variable-list
local-variable-list
free-variable-list
contour-type)

The elements of the declaration are interpreted as follows.

function-type
either SUBR or MSUBR. This field specifies whether the program is to be a function, (*fbpi*), or a LISP macro, (*mbpi*). (Only meaningful for top-level CONTOURs.)

result-type
Specifies the type (ANY, PAIR, NUMBER, etc.) of the value of this program (only meaningful for top-level CONTOURs).

argument-variable-list
The list of identifiers which will be used within the contour defined by this CONTOUR as names for the arguments bound at call-time. These identifiers may be restricted (using the forms defined for LISP), though the restrictions will only be checked on top-level CONTOURs, and they may be marked as FLUID. (Restrictions are not currently implemented.) An example of an *argument-variable-list* is

(A1 (FLUID A2) A3)

local-variable-list
The list of identifiers used to reference locally bound variables (analogous to PROG variables). These, also, may be marked as FLUID. Code is automatically generated to set these variables to NIL upon entry to the contour defined by this CONTOUR.

free-variable-list

A list of variable names used "free" in this contour (i.e. referenced, but not bound). Any variables in the list which are bound by surrounding contours may be marked as REMOTE, however this is not required, as the assembler will detect this case on its own.

contour-type

May be NIL, CLOSED, ENCLOSED, OPEN or an arbitrary non-NIL object. In a top-level contour a value which is not one of the distinguished ids CLOSED, ENCLOSED, OPEN or NIL is forced to CLOSED, internal contours it is forced to ENCLOSED.

A value of NIL causes a flat contour to be defined, with the argument and local variables bound on the existing stack. A flat contour is safe in two cases; if no variables are bound by this CONTOUR; or if no FLUID variables are bound by this CONTOUR and no state saving occurs in this contour, in any enclosed contour, or any function called, at any depth, from this contour or any enclosed contour.

A non-NIL value causes a raised contour to be defined, with the concomitant creation of a new stack-frame.

Values of CLOSED and OPEN are meaningful only in a top level contour. A CLOSED contour, the usual case, is one with a zero ancestor field. This protects the lexical bindings in the calling function from access by this function. A OPEN contour, on the other hand, has access to the lexical bindings in the calling function.

An ENCLOSED contours is used for internal, raised contours and for QUOTEd *bpix*, that is, *bpi* corresponding to LAMBDA expressions occurring as operands in LISP expressions. The code which results from the assembly of an ENCLOSED contour "knows" about the environment in which it will be running, and accesses variables bound in its immediate environment directly rather than via FINDBIND. Such a *bpi* will cause havoc if it is ever run in the wrong environment, and thus it should never be "cut loose" from the FUNARG in which it will be wrapped.

lap-statement

A lap statement may be a

CONTOUR
SECTION
comment
label
pseudo instruction
machine instruction
extended instruction
LAP macro instruction

These will be dealt with below (except, of course, the CONTOUR, which is being dealt with right now). A CONTOUR defines both a new contour and a new level. A contour establishes a new set of variable bindings. It acts as a semi-permeable (one way) barrier to variable references and acts as an impermeable barrier to label references. That is, any variable defined within a contour is unknown to code outside, while a variable defined outside a contour (but within the same LAP program) is known to code within (unless a new variable with the same name is defined on the new contour). A label, however, defined on either side of a contour is unknown to any code on the other side. A level acts as a semi-permeable barrier to label references, that is, any label defined outside a level is accessible to code within that level. In the case of a CONTOUR this is of little interest, as the simultaneously created contour frustrates any outward label references, but an independent level can be established by a SECTION (see the next section).

SECTION

A SECTION has the form

(SECTION lap-instruction*).

A SECTION defines a new level in the program. Labels defined within a SECTION are inaccessible to code outside of the SECTION, but are accessible to code within nested SECTIONs (but not nested CONTOURs). All variables bound by enclosing CONTOURs are accessible, and an SECTION binds no variables itself. Thus, one may branch (TRA) out of a SECTION, but one can only enter a SECTION by "falling into" it, i.e., by encountering it in the flow of control. A SECTION may have a label preceding it, in which case the label is defined in the enclosing level. While it will have the same value as a label occurring as the first item in the SECTION, it will have different scope.

COMMENTS

There are three forms of comments in LAP programs. First, a string ('any characters') is taken as a comment and printed in the listing, offset from the surrounding code by blank lines.
Second, any form with an * as its CAR, i.e. (* any items) will be printed in the listing, with the * appearing in the flag column (see the section on LAP listings).
Finally, any form with a CAR of *** will be printed at the left margin. This is the form used for LAP error messages.

LABELS

Identifiers are taken to be LABELs. They always refer to the next storage-allocating LAP statement to be encountered. Thus, comments occurring between a label and an

instruction will not affect the value of the label. The scope rules for labels are outlined under CONTOUR and SECTION, above.

PSUEDO INSTRUCTIONS

Under the heading of pseudo instructions are grouped two different sets of forms, those which are used to write constants, and those which affect the assembly process. (The distinction between pseudo instructions and extended instructions is hazy at best. Thus, PUSH will generate code but is listed here, while FIXRET may not, but is included with the extended instructions).

Constants

The forms used to write constants are shown in Figure 9. In the (=C *string*) form the characters comprising *string* are laid down in the *bpi* at the current location. If this would cross a page boundary a new page is forced. The maximum length string which can be used is 4090 bytes long.

In the other forms *datum* is evaluated and laid down with the indicated alignment. *datum* may be a number or any of the following:

(LABEL *l*)
: The value will be the location of the label *l*, relative to the beginning of the *bpi*. The scope rules for labels hold.

(DISPLAY . *n*)
: The value will be the location of the display for contour number *n*, relative to the beginning of the *bpi*.

(=>P *progname*)
: The value will be the location of the associated program named *progname*, relative to the beginning of this *bpi*. (see the description of LAP370 arguments).

Once *datum* has been evaluated the right hand "n" bytes are stored in the *bpi*, where "n" is the length given in the table above.

Control instructions

These forms affect the assembly of the program. One of them, USE, will sometimes appear on listings, generated by LAP itself; however, it cannot be used by the programmer, and is only listed here for completeness.

LAP Constant Forms:

Form	Alignment	Length
(=C *string*)	1	(strlength *string*)
(=B *datum*)	1	1
(=H *datum*)	2	2
(=F *datum*)	4	4
(=D *datum*)	4	8
(=V *datum length*)	1	*length*

Figure 9

(EQU *id n*)

Uses of the identifier *id* will be equivalent to the number *n*. This affects op-codes, register names, length codes, and masks in machine instructions. Thus the form

(EQU *S1 14)

would cause *S1 to name register 14 rather than 11. An EQU anywhere in a CONTOUR affects all code directly included or nested in the CONTOUR, unless overridden by another EQU. It will not affect code in an enclosing CONTOUR. If two or more EQUs occur for the same identifier within the same CONTOUR the last one encountered will be the only one effective. Care should be taken in changing op codes, that the EQU precedes any use of the op. If this is not done erroneous lengths may be computed for such instances.

(PUSH [*n*])

PUSH extends the pointer stack *n* (or 1, if *n* is not specified) words, generating code to store NIL in the newly allocated slots. This allocation is static, that is, the location relative to the base of the stack is determined at assembly time, and if the PUSH pseudo instruction occurs in a loop the same location will be used over.

(PUSHA [*id*] [*length* | (*length alignment*)])

PUSHA allocates *length* bytes of space on the abstack, with an alignment of *alignment*. If neither length nor alignment are specified they both default to 4. If alignment is not specified it defaults to 1. If *id* is present it can be used in the address fields of machine instructions to reference the allocated slot in the abstack.

The caveat on the static nature of stack allocation given above holds here as well.

(POP [*n*])
POP de-allocates *n* (or 1) slots on the pointer stack. If more POPs are issued than PUSHs an error message is given, and they are no-oped.

(POPA [*n*])
POPA de-allocates *n* (or 1) slots on the abstack. If any of the "popped" slots have ids associated with them, that association is dropped, and any reference to that id after the POPA is an error. Over popping can occur.

If there are PUSHAs without corresponding POPAs in a CONTOUR, LAP will add the needed POPAs at the end of that CONTOUR, thus preventing references to abstack variable from outside the CONTOUR in which they were established. Reference can be made from within an embedded, flattened, CONTOUR.

MACHINE INSTRUCTIONS

LAP accepts three basic statement forms for machine instructions, with minor variations caused by omission of optional, trailing, arguments.

For RR instructions the form is

(*op* [*r1* [*r2*]])

for RX, RS, SI, and S instructions it is

(*op* [*r1* [*a1* [*x2*]]])

and for SS instructions it is

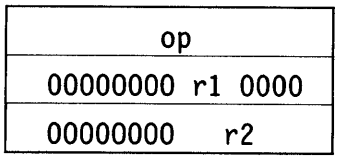
(*op* [*l1* [*a1* [*l2* [*a2*]]]])

During the assembly process (a particularly fitting term in this case) the various fields are evaluated and "or'ed" together as shown in Figure 10.

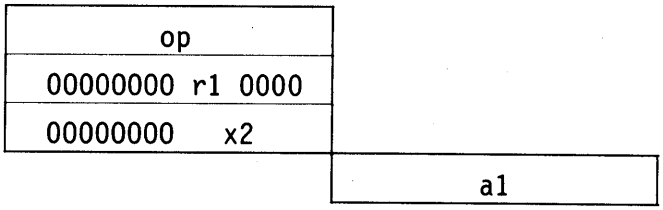
Three types of evaluation are performed, OPVAL, LAPVAL, and RANDVAL. OPVAL is applied to the *op* field; LAPVAL to the *l1*, *l2*, *r1*, *r2* and *x2* fields; and RANDVAL to the *a1* and *a2* fields. After the evaluation the various fields are masked, only the rightmost 4 bits being retained for the *r1* and *l1* fields, the rightmost 8 for the *r2*, *l2* and *x2*, and the rightmost 16 for the *op*, *a1* and *a2*. Any omitted fields are assumed to have a value of zero.

Machine Instruction Composition

RR Instructions



RX, RS, SI, S Instructions



SS Instructions

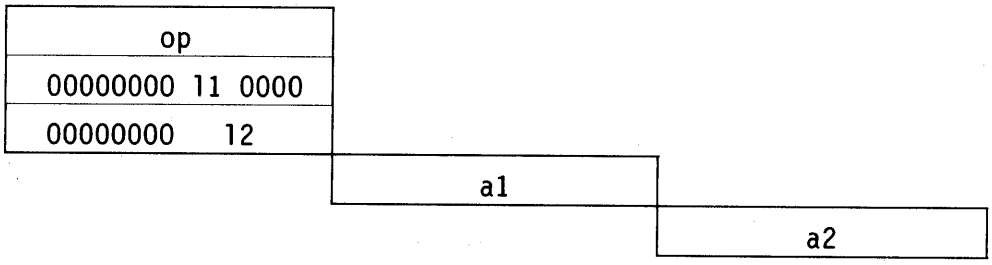


Figure 10

As you may note, this limited set of forms can cause a certain amount of inconvenience. For example, though the I field of an SI instruction may be written as a unit, in the *l2* field, the *l1* field must be present, although with a value of zero in this case.

Thus the instruction

```
TM    X(4),X'37'
```

may be written

```
(TM 0 (4 X) %X37)
```

or, in the older (LAP360) form

```
(TM 3 (4 X) 7)
```

As LAP is primarily produced by the LISP compiler, rather than by humans, this is felt to be a price which can be paid, to buy a simpler format.

LAPVAL

We will describe LAPVAL first, as both OPVAL and RANDVAL may default to LAPVAL in certain cases. We will list the forms for which LAPVAL is defined, together with their values.

numbers
: Numbers have themselves as LAP values.

identifiers
: If an identifier has been given a value by an EQU pseudo instruction or by an initial symbol table (see LAP370 arguments, below) that will be its LAP value. Otherwise we check for an INTSYM property on the identifier, and if it is present, its value becomes the LAPVAL. If no such property exists, the value is zero.

(- *item*)
: The LAPVAL is the negation (that is, (- *item*) + *item* = 0) of the LAPVAL of *item*.

(LABEL *label*)
: If we are in the scope of definition of *label*, the LAPVAL is the location defined by that label, relative to the beginning of the *bpi*.

OPVAL

To find the OPVAL of an item (used for *op* fields only) we first check for a value defined by an EQU or an initial symbol table. If none exists we then check for the property OPSYM, returning its value, if any. If no such property exists we default to the S/370 machine code, which is built in, as a table, in the OPVAL

function. The final result, what ever its provenance, is masked to its rightmost sixteen bits.

RANDVAL

The address field(s) of the instruction is evaluated by RANDVAL. It can have any one of a number of forms.

(*b d*)
: This is a base, displacement pair. The LAPVALs of *b* and *d* are found and that of *b* is multiplied by 4096 (to shift it to the base position in the address half word of the instruction) and added to the LAPVAL of *d*.

(*b d* [*inc**])
: This is simply an extension of the previous case. Here the displacement is computed by summing the LAPVALs of *d* and the various *inc*s.

identifier
: *identifier* must be a variable known in the current scope, otherwise this is treated as an error. A reference to the variable will be constructed, with the proper index depending on whether *identifier* is bound on the *hE*, the stack, or is defined on the abstack. If *identifier* is a FREE variable an indirect reference is constructed, via a "foul" pointer in the stack, and the necessary additional code to establish addressability is inserted into the LAP program. Register *SCRATCH will be used.

(HEAD *identifier*)
(FLUID *identifier*)
: If *identifier* is bound in the *hE*, this is the same as *identifier*, otherwise it is an error. HEAD is used for *lexical* bindings, while FLUID is used for *fluid* bindings.

(STACK *identifier*)
(ASTACK *identifier*)
(FREE *identifier*)
: As above.

(INDIRECT *identifier*)
: This form evaluates to a reference to the "foul" pointer which, in turn, points to the current binding of *identifier*. This is the first half of the indirect reference to a FREE variable. For example, the form

```
(L *S1 X)
```

where X is a *fluid* variable being used FREE, would expand into

```
(L *SCRATCH (INDIRECT X))
(L *S1 (*SCRATCH 0))
```

(PUSH)

Causes LAP to allocate a new slot on the pointer stack, and assembles to a reference to that slot. This form is only proper in full word store, or four byte move, instructions. In any other context the execution of the instruction will have unpredictable, even catastrophic results. No checking for proper usage is done by LAP.

(POP)

Assembles to a reference to the last slot allocated on the pointer stack, and de-allocates same. Once a datum has been fetched from the stack or examined on the stack using a POP it is effectively lost (unless retained in a register).

(TOP [*n* [*inc**]])

(TOP) allows references to items in the pointer stack without changing its status. The form (TOP) refers to the last item PUSHed onto the stack. (TOP *n*) refers to the *n*th item back on the stack, thus (TOP 0) is equivalent to (TOP). Any increment is added to the displacement of the address. Thus (TOP 0 3) would address the rightmost byte of the last pointer PUSHed on the stack.

(BASE [*n* [*inc**]])

Like TOP, but the indexes from the first item pushed on the stack, rather than the last. Note that the first item(s) on the stack may be saved shallow bindings or dump/foul pointer pairs, for free references, so beware.

(PUSHA [*id*] [*l* | (*l a*)])

This acts the same as the PUSHA pseudo instruction, while assembling into the address of the newly created abstack entry.

(POPA)

Assembles to an access to the last item PUSHAed onto the abstack, while deallocating the item. Note that only one item may be dropped at a time when using POPA as an address field rather than as a pseudo instruction.

(TOPA [*n* [*inc*]])

Same as (TOP) but refers to items on the abstack. This can be used if identifiers were not attached to the items. (TOP *n*) adjusts for gaps in the abstack caused by alignment.

(LABEL *label*)

Assembles to a reference to the label *label*, if it is both within the scope of the current context and accessible. Any label defined on the zeroth page of the *bpi* is accessible from anywhere in the *bpi*, while a label defined on any other page of the *bpi* is accessible only from that same page. LAP assumes, for labels not on the zeroth page, that *RET contains the page base. Any other case is considered an error.

(=>T *label*)

If *label* is on the same page and at a higher address than the instruction containing this form, it is equivalent to (LABEL *label*). Otherwise it assembles to an entry in the trampoline area at the foot of the current page. For off-page branches, the trampoline area contains code which establishes addressability on the new page, and an unconditional branch to the label *label*. For "upward" branches (i. E. Branches to lower addresses, whether on the same page or not) code is included to make interrupt servicing possible. Thus code using =>T branches will contain no uninterruptible loops. Use of the form (=>T *label*) in any but a branch instruction will have unexpected results.

(EXTERNAL *id*)

The identifier *id* is checked for an EXSYM property. if such exists it is returned as the value of the address field, otherwise an error is indicated. The value of the EXSYM is expected to be a sixteen bit number, with the proper base register bits in its high four bits. The base will normally be either *FIX or *NIL.

(=C *item*)
(=B *item*)
(=H *item*)
(=F *item*)
(=D *item*)
(=V *item n*)

Causes a constant (see pseudo instructions) to be assembled at the foot of the current page, and assembles a reference to it as the address of the instruction. Literals are shared whenever possible. The same rules for accessibility apply as hold for labels.

(QUOTE *s-expression*)

Causes *s-expression* to be constructed in the heap, with a communication cell pointing to it, and assembles the code needed to reference that cell. The code needed will vary, depending on the instruction in which the QUOTE occurs, see =>Q below. Use of (QUOTE ...) in any instruction which modifies storage will cause unpredictable results.

(=>Q *s-expression*)

Causes a word to be assembled, at the foot of the page, containing the address, relative to *NIL, of the communication cell pointing to *s-expression*. If *s-expression* is a small integer, the small integer itself is assembled at the foot of the current page, and a direct reference to it is given as the address. =>Q is used in the expansions of instructions containing (QUOTE ...) forms. RX instructions having no explicit X fields, such as

```
(L *S1 (QUOTE sexp))
```

become

```
(L *SCRATCH (=>Q sexp))
(L *S1 (*SCRATCH 0) *NIL)
```

SS instructions use the same logic as RX instructions with X fields, thus

```
(MVC 0 (PUSH) 4 (QUOTE sexp))
```

becomes

```
(L *SCRATCH (=>Q sexp))
(ALR *SCRATCH *NIL)
(MVC 0 (PUSH) 4 (*SCRATCH 0))
```

(=>FL *id**)

Causes a cell (or cells) to be assembled at the foot of the current page with a pointer(s), relative to *NIL, to the shallow binding cell(s) of the variable(s) *id*.

(=>FN *fn* ([*type**]) *type*)

Assembles two contiguous words at the foot of the current page containing the offset, relative to *NIL, of the value cell (shallow binding cell) pointing to the function *fn*, and the argument type code corresponding to the *types* in the form. The address assembled is that of the first word.

(*b* (HEADP *id*))
(*b* (STACKP *id*))
(*b* (INDIRECTP *id*))
Similar to (HEAD *id*), etc., but uses the base register specified by *b*.

EXTENDED INSTRUCTIONS

There are a number of forms acceptable to LAP which are not S/370 machine instructions nor assembly control (pseudo) instructions. In form and behavior they resemble LAP macros but they are built into the assembler rather than being defined as LISP functions. They are grouped here for convenience.

(TRA *cond label*)
TRA expands into a branch instruction, with an *a* field (=>T *label*). The *cond* can be a number, the letter U, or any letter or combination of letters which, when prefaced by B, form a legal LAP branch instruction. If *cond* if a number a BC is produced, while a U results in an unconditional branch.

(LOAD *r1 a1*)
In most cases a LOAD will simply be transformed into the corresponding L instruction. However the *a1* field is allowed to have one form which is not allowed in a machine instruction, that of a CONTOUR. If the *a1* field is a CONTOUR it is assembled separately, with its display chained to the program currently being assembled, and the value loaded will be a pointer to the *bpi* so produced. This form is used for LAMBDAs in operand position.

(STORE *r1 a1*)
Exactly equivalent to

(ST *r1 a1*)

In the future both the STORE and the LOAD may be given more "intelligence", to allow them to cope with floating point numbers, half word values, and/or other data types.

(RETURN)
Assembles into the code needed to exit from the current contour. It is up to the programmer to provide a value in register *S1.

(CALL *fn* (*arg-type**) *res-type*)
Assembles as the code needed to call the function *fn*. The type checking code is derived from the *arg-type*(s) and the *res-type* (result type).

(CALLAC (*arg-type*+) *res-type*)

Assembles into the code to apply the object pointed to by *S1 to the last n (n = number of *arg-type* items) values PUSHed on the stack.

(SKPNXT *cond*)

Assembles into code to skip the next LAP instruction (machine or extended) if the condition *cond* holds. *cond* may be any value appropriate to the cond field of a TRA extended instruction.

(SKPTRU *cond*)

equivalent to

```
(TRA cond label)
(LOAD *S1 (EXTERNAL TRUE))
label
```

(SKPNIL *cond*)

equivalent to

```
(TRA cond label)
(LR *S1 *NIL)
label
```

(FIXRET)

On the zeroth page of a program, FIXRET is a no-op. On subsequent pages it must follow a

```
(BAL *RET ... )
```

or a

```
(BALR *RET ... )
```

It assembles as code to reset the *RET register to the beginning of the current page. This allows the dual use of *RET, as both a base register and a function linkage register.

(GOIF [*tmpreg*] *var type label*)
(GOIFNOT [*tmpreg*] *var type label*)
(GOIFR *reg type label*)
(GOIFNOTR *reg type label*)

These are conditional branches, based on the LISP type of an argument. In the case of GOIF and GOIFNOT the argument is a variable *var*, in storage. In the case of GOIFR and GOIFNOTR the argument is the contents of the register *reg*. The optional first argument in the storage forms is a register to be used, if needed, for the type test. If this is omitted *SCRATCH will be used. The types which can be tested are:

ATOM	LIST	PLINT
BVEC	MKID	RVEC
FIX	MR	SMINT
FLOAT	MSUBR	STATE
FR	NLINT	STREAM
FVEC	NTUPLE	STRING
GENSYM	NULL	SUBR
IDENT	NUMBER	VEC
IVEC	PAIR	
LINT	PLEX	

Thus, to branch to the label FLOATING if the variable X is a floating point number, one would write

```
(GOIF X FLOAT FLOATING)
```

while to branch to the label ELEMENT if register *S3 did not contain a pair pointer one would write

```
(GOIFNOTR *S3 PAIR ELEMENT)
```

If the *type* field is a number rather than one of the key words listed above the low order eight bits of the number will be compared with the type byte of the argument, and the branch taken accordingly. Thus to test for a rational number one could write

```
(GOIF Y %XB0 RATIONAL)
```

(LOADVECL *reg1 var*)
(LOADVECLR *reg1 reg2*)

These forms generate code to load the register *reg1* with the length (as a S/370 integer, not a LISP number) of the vector pointed to by the second argument. The length code type byte will be deleted.

(RECLAIM)
Generates the code to cause a garbage collection.

(POLL)
Generates code to poll for an interrupt.

(LERROR)
Generates code to enter the error break loop with an error-channel of 12 (return expected) and with the current contents of *S1 bound to ?ARGS?.

(SAVESTATE)
Generates code to save the current state and load *S1 with a pointer to the resulting SD.

(SETFRT)
Generates code to make *FRT correct.

LAP MACRO INSTRUCTIONS

As each form is processed by LAP its CAR is examined. If the CAR is and identifier which is bound in the current EXPANSIONSTATE it is evaluated. If the value is a LLAMBDA form it is APPLYed to the original form, and the value returned is "spliced" into the LAP program in place of the form. The value of the macro must be a list of LAP statements. It may include labels as well as other LAP macros. (Note that pseudo instructions and extended instructions are processed before LAP macros, and thus may not be redefined.)

APPENDIX 1

LISP/370 in the TSO Environment

Information about the operation of LISP/370 under TSO is presented in this section. Information is gathered in this one place in order to provide a more coherent description than would be possible with various details scattered about in this publication.

Unlike CMS, where the user is a world unto himself in his own virtual machine, the TSO user operates in a larger environment which he shares with other TSO users and MVS batch operations. This larger environment usually needs more in the way of operating conventions and procedures to which the local user population must adhere in order to preserve order and communicate among themselves. To run LISP in this environment, the user must know something about how LISP input/output operates in the TSO environment, and the relationship between LISP and the rest of TSO. The remainder of this section discusses these items, and defines how several LISP functions cannot perform in the normal manner in this environment.

There are versions of LISPGET, STRTLISP and LISPFREE programs written specifically for the TSO environment. These reside in an MVS load library called LISP.LOAD (the name of the library is at the user's discretion; this name is used here). This dataset must be allocated with ddname LISPLOAD. The user must allocate the desired LISP/370 file image dataset with ddname FILEIM, then invoke the LISPGET program. This program processes parameters which may be supplied to designate the desired allocation of space to various LISP purposes, then obtains a large element of storage in subpool 78. This subpool is shared by the TSO command processor with the programs it invokes as subtasks (LISPGET in this case). LISPGET bootstraps the LISP file image into this area, and passes control to it to complete reading of the file image file and start LISP. When loading the file image distributed as part of LISP/370, the function SUPV will be invoked with input and output streams defined for the user's console, and on initial entry will write the message: LISP.

To return to the TSO environment from the LISP environment, use the function application (RET). Because LISP has been loaded into shared storage, it will remain there until it is explicitly freed by the program LISPFREE. To resume execution in the previously loaded LISP system, use the STRTLISP program. Thus a typical terminal session might go like this:

```
ALLOC FI(FILEIM) DA(LISP370.FILEIM) SHR
ALLOC FI(LISPLOAD) DA(LISP.LOAD) SHR
CALL LISP.LOAD(LISPGET)
    ...
    ... (LISP expressions read,
    ... evaluated, and printed by SUPV)
    ...
(RET)
    ...
    ... TSO commands such as
    ... file allocation or editing
    ...
CALL LISP.LOAD(STRTLISP)
    ...
    ... (More LISP expressions)
    ...
(RET)
CALL LISP.LOAD(FREELISP)
```

You will probably wish to define TSO CLIST's to perform these functions, so that the entire allocation and invocation of LISPGET might be achieved by use of the single command LISP, et cetera.

The options which may be specified for LISPGET are discussed in the sectiion titled "How to Access LISP/370". For TSO, the CMSHIGH= parameter is not recognized, and no space is used between the parameter names and values. Thus, to allocate 2 megabytes for running LISP, and specify that 400K bytes of storage be reserved for additional binary programs, the appropriate command is:

```
CALL LISP.LOAD(LISPGET) '2M,BPI=400K'
```

A note about the Structured Programming Facility. SPF does not share subpool 78 with subtasks which it attaches for invoking TSO commands. Therefore, if LISPGET is invoked from SPF, the LISP system loaded will be lost when a return is made to SPF. LISP may be loaded first, then (RET) used to return control to TSO, where SPF may then be initiated. Then it will be possible to reenter LISP when an exit is made from SPF.

Allocation of datasets in TSO may be a complex process, depending upon the nature of the dataset involved and the device(s) on which it is located. To avoid this complexity as much as possible, LISP does not perform dynamic allocation of datasets. Rather, the procedure described above for exiting to TSO then reentering LISP makes it possible for the user to take advantage of all of the commands available for dataset allocation in the TSO environment, plus the extensive error diagnostic and informational facilities also available there.

LISP/370 streams defined on files give the user direct access to records in those files, in whatever order he chooses, not necessarily sequential. This ability to directly access data records is used principally in processing LISPLIB files, but is present for other types of datasets too. To support this direct access to data records, LISP files are stored as VSAM

keyed datasets. This has the additional advantage of making the structure of the files device independent, and giving the user a large variety of commands (through Access Method Services) for defining and manipulating these files.

The MAKEKEY2 utility, supplied with LISP/370, will copy a sequential file into an indexed VSAM cluster while generating appropriate keys. Thus, suitable models for LISP VSAM datasets might be defined by commands such as these:

```
DEFINE CLUSTER( NAME(MODEL.LISP370) VOLUMES(MVS278) -
   INDEXED KEYS(4 0) RECORDS(100 200) RECORDSIZE(60 256) -
   SPEED)

DEFINE CLUSTER( NAME(MODEL.LISPLIB) VOLUMES(MVS278) -
   INDEXED KEYS(4 0) RECORDS(100 200) RECORDSIZE(80 80) -
   SPEED)
```

Then, if a user has a sequential dataset named IT.FUNCTION, he could define and load it into a VSAM cluster with the commands:

```
DEFINE CLUSTER( NAME(IT.LISP370) MODEL(MODEL.LISP370))
ALLOC FI(KEY) DA(IT.LISP370) OLD
ALLOC FI(SEQ) DA(IT.LISP370) SHR
CALL LISP.LOAD(MAKEKEY2)
```

All of the routines which interface LISP with MVS/TSO are in the module LISPVS. This module is separately generated and, should the user have a need for his own system-dependent functions, he may modify or add code to LISPVS and replace this module in LISP.LOAD. Since LISPVS is dynamically loaded by LISPGET, it may be changed independently of storing a new LISP file image. With the exception of LISPVS, all programs and data active in the LISP system will be written by (FILELISP) into the file image file so that they are recovered by the next LISPGET referencing that file.

LISP file images are sequential files created by the LISP FILELISP function. This function uses ddname FILELISP to define an appropriate output dataset. This dataset must have F-format, 800-byte records, blocked as the user wishes. To change the output dataset for a LISP file image, leave LISP using (RET), allocate ddname FILELISP as desired, then return to LISP and issue (FILEIM).

LISPVS will dynamically match dataset names provided by LISP functions with ddnames currently defined. Except for the ddnames mentioned explicitly above, the actual ddname used has no significance. Since TSO installations often have certain conventions regarding composition of dataset names, this dsname-to-ddname process will succeed even if given only partial dsnames. The match algorithm takes the dsname qualifiers in the order they are given (in the FILE attribute of a call to DEFIOSTREAM, for example), and will select a ddname for any dataset which matches the given qualifiers with zero or more additional, prefix dsname qualifiers. For example, if the argument (FILE X LISP370) is given to DEFIOSTREAM, a match will be found for a ddname allocated to the dataset RYNIKER.V.X.LISP370. The first acceptable ddname found is used. Avoidance of ambiguities is the responsibility of the user.

One of the benefits obtained by this partial matching scheme is that LISP programs may be written to operate with certain datasets independent of the userid prefix commonly used by TSO. The presence (or absence) of the qualifier RYNIKER in the above example is not relevent.

In an effort to make it easier to write LISP functions which can operate in a similar manner in both TSO and CMS environments, another abbreviation mechanism is implemented in the dsname-ddname matcher. If exactly three dsname qualifiers are supplied by the user's program, and the third has a valid form for a CMS filemode (i.e. it is either *, or a single capital letter, or a capital letter followed by a digit from 0 to 5), a match will be declared if a ddname is found allocated to a dsname meeting the above matching algorithm without the final qualifier. For instance, the above example would also succeed if the argument value was (FILE X LISP370 A1).

Because LISP does not perform dynamic allocation in TSO, the functions IOSTATE, IOSTATEW, and ERASE are effectively constants in the TSO environment. They have no explicit side effects and return the value (). These functions do, however, cause control to pass into the system interface module LISPVS, so they could be given meaning by local modification of that program.

OBEY also is inhibited in the TSO environment. Its function can be achieved by exiting to TSO through (RET), then reentering LISP with STRTLISP.

FILELISP always uses ddname FILELISP for its output file. Dataset name will be whatever was specified in the allocation for that ddname. The argument(s) of FILELISP are not relevent.

Printed below is the contents of the first file on the distribution tape. This JCL defines the contents and format of the tape, and provides an outline of the steps required to retrieve information from the tape. Only the contents of files two and three on the distribution tape are required for running LISP under TSO. The other files contain source and documentary materials. It is recommended that the programmer installing LISP generate a source library from the fourth file, since often questions regarding the detailed behavior of a LISP function can be answered by a brief examination of its definition. Generation of this library may be done at the user's convenience, however. It is not necessary to have this available before running LISP.

```
//* SAMPLE JCL TO READ LISP/370 DISTRIBUTION TAPE.
//*
//* FILE 1 IS COPYRIGHT NOTICE.
//*
//*  FILE 2 IS JCL DEFINING DISTRIBUTION TAPE.
//*
//*  NOTE:  THE PURPOSE OF THIS IS PRIMARILY DESCRIPTIVE.
//*         ADJUSTMENTS WILL HAVE TO BE MADE TO REFLECT
```

```
//*       THE INDIVIDUAL CHARACTERISTICS OF THE USER'S
//*       INSTALLATION.
//*
//FILE2 EXEC PGM=IEBGENER
//*
//SYSPRINT DD SYSOUT=A
//SYSUT1 DD UNIT=TAPE,DISP=(OLD,PASS),VOL=(PRIVATE,RETAIN,SER=XXXX),
//  LABEL=(2,NL),DCB=(LRECL=80,RECFM=FB,BLKSIZE=14400,DEN=4)
//SYSUT2 DD DSN=LISP.JCL,UNIT=SYSDA,DISP=(,CATLG),
//  DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120),
//  SPACE=(TRK,(1,1))
//SYSIN DD DUMMY
//*
//*
//* FILE 3 CONTAINS LINK EDITOR INPUT TO CREATE LISP LOAD LIBRARY.
//*
//FILE3 EXEC PGM=IEWL,PARM='LIST'
//SYSPRINT DD SYSOUT=A
//SYSUT1 DD UNIT=SYSDA,SPACE=(TRK,(10,10))
//SYSLMOD DD UNIT=SYSDA,DSN=LISP.LOAD,SPACE=(TRK,(10,5,13)),
//  DISP=(,CATLG)
//SYSLIN DD VOL=(PRIVATE,RETAIN,REF=*.FILE2.SYSUT1),LABEL=(3,NL),
//  DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200),DISP=(OLD,PASS)
//*
//*
//* FILE 4 CONTAINS A LISP370 FILE IMAGE.
//*
//FILE4 EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=A
//SYSUT1 DD VOL=(PRIVATE,RETAIN,REF=*.FILE2.SYSUT1),LABEL=(4,NL),
//  DCB=(LRECL=800,RECFM=FB,BLKSIZE=12800),DISP=(OLD,PASS)
//SYSUT2 DD UNIT=SYSDA,DSN=LISP370.FILEIM,DISP=(,CATLG),
//  SPACE=(800,(750,50)),DCB=(LRECL=800,RECFM=FB,BLKSIZE=12800)
//SYSIN DD DUMMY
//*
//*
//*  THAT'S THE END OF THE DATA REQUIRED TO RUN LISP/370.
//*  THE REMAINING FILES CONTAIN OPTIONAL MATERIALS
//*  SUCH AS SOURCE PROGRAMS AND LISTINGS.
//*
//*
//*  FILE 5 CONTAINS SOURCE CODE FOR PROGRAMS CODED IN
//*  EITHER LISP OR LAP.  MAKING A PARTITIONED DATASET
//*  CONTAINING V-FORMAT RECORDS PRESENTS SOME PROBLEMS
//*  BECAUSE WE DON'T HAVE A PRECISELY RIGHT UTILITY
//*  PROGRAM.  THEREFORE, THE V-FORMAT RECORDS HAVE BEEN
//*  PACKED INTO 80-BYTE FIXED-FORMAT RECORDS, AND THE
//*  COPYV PROGRAM (IN LISP.LOAD) WILL RECONSTRUCT THEM
```

```
//*  INTO THE ORIGINAL V-FORMAT RECORDS.
//*     A SAMPLE JOB STEP TO PRINT ONE MEMBER OF THIS
//*     LIBRARY APPEARS BELOW WITH THE STEP NAME COPYV.
//*
//FILE5 EXEC PGM=IEBUPDTE,PARM='NEW'
//SYSPRINT DD DUMMY
//SYSUT2 DD DSN=LISP370.SOURCE,UNIT=SYSDA,
//  SPACE=(3120,(360,30,30,15),,,ROUND),
//  DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120),DISP=(,CATLG)
//SYSIN DD VOL=(PRIVATE,RETAIN,REF=*.FILE2.SYSUT1),LABEL=(5,NL),
//  DCB=(LRECL=80,BLKSIZE=14400,RECFM=FB),DISP=(OLD,PASS)
//*
//*
//*  FILE 6 CONTAINS INPUT TO CREATE LIBRARY OF LISPLIB FILES.
//*  THESE WILL HAVE TO BE PROCESSED BY MAKEKEY2 BEFORE THEY
//*  CAN BE READ BY A LISP PROGRAM.
//*
//FILE6 EXEC PGM=IEBUPDTE,PARM='NEW'
//SYSPRINT DD DUMMY
//SYSUT2 DD UNIT=SYSDA,DSN=LISP370.LISPLIB,
//  SPACE=(3120,(334,30,30,15),,,ROUND),
//  DCB=(LRECL=80,RECFM=FB,BLKSIZE=3120),DISP=(,CATLG)
//SYSIN DD VOL=(PRIVATE,RETAIN,REF=*.FILE2.SYSUT1),LABEL=(6,NL),
//  DCB=(LRECL=80,RECFM=FB,BLKSIZE=14400),DISP=(OLD,PASS)
//*
//*
//*  FILE 7 CONTAINS INPUT DATA TO CREATE LISP MACRO LIBRARY.
//*  THIS LIBRARY IS REQUIRED FOR ASSEMBLING THE MODULES
//*  IN LISP.LOAD.
//*
//FILE7 EXEC PGM=IEBUPDTE,PARM='NEW'
//SYSPRINT DD DUMMY
//SYSUT2 DD DSN=LISP.MACLIB,DISP=(,CATLG),
//  SPACE=(3120,(120,10,15),,,ROUND),
//  DCB=SYS1.MACLIB,UNIT=SYSDA
//SYSIN DD VOL=(PRIVATE,RETAIN,REF=*.FILE2.SYSUT1),LABEL=(7,NL),
//  DCB=(LRECL=80,BLKSIZE=14400,RECFM=FB),DISP=(OLD,PASS)
//*
//*
//*  FILE 8 CONTAINS DIRECTORY/CROSS-REFERENCE OF LISP FUNCTIONS.
//*  THESE RECORDS DO NOT CONTAIN PRINTER CARRIAGE CONTROL
//*  INFORMATION, BUT THEY DO CONTAIN SOME LOWER CASE CHARACTERS
//*  AND SO SHOULD BE PRINTED ON AN APPROPRIATE PRINTER.
//*
//FILE8 EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=A
//SYSUT1 DD VOL=(PRIVATE,RETAIN,REF=*.FILE2.SYSUT1),LABEL=(8,NL),
//  DCB=(LRECL=84,RECFM=VB,BLKSIZE=3156),DISP=(OLD,PASS)
```

```
//SYSUT2 DD SYSOUT=A
//SYSIN DD DUMMY
//*
//*
//*  FILE 9 CONTAINS THE SOURCE CODE FOR ALL OF THE ASSEMBLY
//*  LANGUAGE MODULES SUPPLIED WITH LISP/370.
//*     AN EXAMPLE OF THE JCL NEEDED TO ASSEMBLE THE LISP370
//*     MODULE (THE ONLY ONE WHICH REQUIRES MORE THAN THE
//*     TRIVIAL ONE-MODULE INPUT, SIMPLE LINKEDIT) APPEARS
//*     LATER WITH STEP NAME LISPASM.
//*  ASSEMBLY OF THE LISP370 MODULE REQUIRES THE PROGRAM PRODUCT
//*  ASSEMBLER-H.  THE OTHER MODULES NECESSARY TO OPERATING LISP
//*  (LISPGET, LISPFREE, AND STRTLISP) MAY BE ASSEMBLED WITH
//*  ASSEMBLER-F.
//*
//FILE9 EXEC PGM=IEBUPDTE,PARM='NEW'
//SYSPRINT DD DUMMY
//SYSIN DD VOL=(PRIVATE,RETAIN,REF=*.FILE2.SYSUT1),LABEL=(9,NL),
//  DCB=(RECFM=FB,LRECL=80,BLKSIZE=14400),DISP=(OLD,PASS)
//SYSUT2 DD DSN=LISP370.ASM,DISP=(,CATLG),UNIT=SYSDA,
//  SPACE=(3120,(1080,60,15),,,ROUND),
//  DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120)
//*
//*
//*  FILE 10 CONTAINS THE LISTING FILE PRODUCED BY AN ASSEMBLY OF
//*  THE LISP370 NUCLEUS.
//*
//FILE10 EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=A
//SYSUT1 DD VOL=(PRIVATE,REF=*.FILE2.SYSUT1),LABEL=(10,NL),
//  DCB=(RECFM=FBM,LRECL=121,BLKSIZE=12100),DISP=OLD
//SYSUT2 DD SYSOUT=A
//SYSIN DD DUMMY
//*
//*
//*  THE REMAINING FILES ON THE DISTRIBUTION TAPE CONTAIN SIMILAR
//*  DATA IN A FORMAT SUITABLE FOR THE VM/370-CMS USER.
//*
//*
//*  TO RUN THE COPYV PROGRAM TO GENERATE A LISP SOURCE MODULE.
//*
//COPYV EXEC PGM=COPYV
//STEPLIB DD DSN=LISP.LOAD,DISP=SHR
//SYSIN DD DSN=LISP370.SOURCE(DEFINE),DISP=SHR
//PRINT DD UNIT=SYSDA,DSN=&&COPY,DISP=(,PASS),
//  SPACE=(3120,(20,10),,,ROUND),
//  DCB=(RECFM=VB,LRECL=137,BLKSIZE=3120)
//*
```

```
//*
//*  TO DEFINE VSAM DATASETS FOR SOURCE AND LISPLIB FILES.
//*  NOTE:  THIS IS A MINIMAL DEFINITION... BETTER PERFORMANCE
//*  COULD UNDOUBTEDLY BE OBTAINED BY EXPERT SPECIFICATION OF
//*  THE VARIOUS VSAM PARAMETERS TO MATCH THE CHARACTERISTICS OF
//*  THE OPERATING SYSTEM.
//*
//*     NOTE THAT ONCE THE MODELS ARE DEFINED, IT IS SIMPLE TO
//*     DEFINE ADDITIONAL CLUSTERS USING THE MODEL.
//*
//AMS EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
   DEFINE CLUSTER(NAME(V.MODEL.LISP370) VOLUMES(MVS278) -
      INDEXED KEYS(4 0) RECORDS(400 400) RECORDSIZE(60 256) -
      SPEED)
   DEFINE CLUSTER(NAME(V.MODEL.LISPLIB) VOLUMES(MVS278) -
      INDEXED KEYS(4 0) RECORDS(400 200) RECORDSIZE(80 80) -
      SPEED)
   DEFINE CLUSTER(NAME(V.DEFINE.LISP370) MODEL(V.MODEL.LISP370))
   DEFINE CLUSTER(NAME(V.DEFINE.LISPLIB) MODEL(V.MODEL.LISPLIB))
/*
//*
//*
//*  NOW FOR A SAMPLE USE OF MAKEKEY2.
//*
//MAKEKEY EXEC PGM=MAKEKEY2
//STEPLIB DD DSN=LISP.LOAD,DISP=SHR
//SEQ DD DSN=&&COPY,DISP=(OLD,DELETE)
//KEY DD DSN=V.DEFINE.LISP370,DISP=OLD
//*
//*
//*  TO ASSEMBLE THE LISP NUCLEUS.
//*  (PROPER LINKEDIT PROCEDURE IS SHOWN IN FILE 2.)
//*
//LISPASM EXEC ASH,PARM.C=('XREF(FULL)','FLAG(5)',
//  NODECK,ESD,NORLD,OBJECT,'SYSPARM=MVS')
//C.SYSLIB DD
//  DD DSN=LISP.MACLIB,DISP=SHR
//C.SYSLIN DD DSN=LISP.OBJ(LISP370),DISP=(,CATLG),UNIT=SYSDA,
//  SPACE=(3120,(64,10,10),,,ROUND),
//  DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120)
//C.SYSIN DD DSN=LISP370.ASM(DUMMYSEC),DISP=SHR
//  DD DSN=LISP370.ASM(LLOADER),DISP=SHR
//  DD DSN=LISP370.ASM(SYSDEPV),DISP=SHR
//  DD DSN=LISP370.ASM(NILSEC),DISP=SHR
//  DD DSN=LISP370.ASM(FIXED1),DISP=SHR
//  DD DSN=LISP370.ASM(FIXED2),DISP=SHR
```

```
//  DD DSN=LISP370.ASM(FIXED3),DISP=SHR
//  DD DSN=LISP370.ASM(FIXED3$),DISP=SHR
//  DD DSN=LISP370.ASM(FIXED3A),DISP=SHR
//  DD DSN=LISP370.ASM(FIXED3B),DISP=SHR
//  DD DSN=LISP370.ASM(FIXED4),DISP=SHR
//  DD DSN=LISP370.ASM(RECLAIM),DISP=SHR
//  DD DSN=LISP370.ASM(BPISEC),DISP=SHR
//  DD DSN=LISP370.ASM(LOADVOL),DISP=SHR
//  DD DSN=LISP370.ASM(STACK),DISP=SHR
//  DD DSN=LISP370.ASM(HEAPSEC),DISP=SHR
//  DD DSN=LISP370.ASM(STORAGE),DISP=SHR
//*
//*
//
```

Index to Function Descriptions

Note: the notation (system function) after a function name indicates a function which is considered of no general interest, or which requires some special knowledge about the details of LISP/370 implementation in order to be correctly used. These functions are not documented in this publication but are included in this index where there is a possibility they might be inadvertently redefined by the general user.

An attempt has been made to adhere to a naming convention allowing the casual user to avoid problems involving his names conflicting with the names of system functions or system variables. This convention is that the names of system functions will include a comma, thereby allowing the user to concoct arbitrary names which do not contain commas and also do not appear in this index.

LISP/370 Program Description/Operations Manual Printed in U.S.A. SH20-2076-0

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601

LISP/370
Program Description/Operations Manual

SH20-2076-0

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM shall have the nonexclusive right, in its discretion, to use and distribute all submitted information, in any form, for any and all purposes, without obligation of any kind to the submitter. Your interest is appreciated.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name and mailing address:

_____

_____

_____

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

(Optional Wording)

What is your occupation? _____

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Reader's Comment Form

Fold and tape | Please Do Not Staple | Fold and tape

First Class
Permit 40
Armonk
New York

Business Reply Mail

No postage stamp necessary if mailed in the U.S.A.

Postage will be paid by:

International Business Machines Corporation
Department 825
1133 Westchester Avenue
White Plains, New York 10604

Fold and tape | Please Do Not Staple | Fold and tape

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601

Cut or Fold Along Line

LISP/370 Program Description/Operations Manual Printed in U.S.A. SH20-2076-0

LISP/370
Program Description/Operations Manual

SH20-2076-0

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM shall have the nonexclusive right, in its discretion, to use and distribute all submitted information, in any form, for any and all purposes, without obligation of any kind to the submitter. Your interest is appreciated.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name and mailing address:

_____

_____

_____

(Optional Wording)

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

What is your occupation? _____

Number of latest Newsletter associated with this publication: _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Reader's Comment Form

Fold and tape | Please Do Not Staple | Fold and tape

First Class
Permit 40
Armonk
New York

Business Reply Mail

No postage stamp necessary if mailed in the U.S.A.

Postage will be paid by:

International Business Machines Corporation
Department 825
1133 Westchester Avenue
White Plains, New York 10604

Fold and tape | Please Do Not Staple | Fold and tape

Cut or Fold Along Line

LISP/370 Program Description/Operations Manual Printed in U.S.A. SH20-2076-0

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601