This is an informal report intended primarily for internal or limited external distribution. The opinions and conclusions stated are those of the author and may or may not be those of the laboratory.

UCID - 16650

LAWRENCE LIVERMORE LABORATORY

University of California/Livermore, California

UT LISP

REFERENCE MANUAL

LLL EDITION
(Preliminary)

E. M. Greenawalt

August 1, 1973

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

MASTER

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

CL 210

UT LISP REFERENCE MANUAL

LLL EDITION

+++++

LLL now possesses a working implementation of UT LISP for the 7600's. UT LISP was originated and developed on CDC 6000 series equipment at the University of Texas at Austin and is now considered the standard LISP implementation on 6000 equipment all over the world.

The enclosed pages are preliminary versions of chapters of a reference manual now in preparation at the University of Texas. Some editing has been done to make them conform to usage at LLL. Although the reference manual is incomplete at the time of this writing, the information content is complete and accurate for use at LLL. Those chapters missing primarily cover material that is highly system-dependent and relevant only to the 6000 version.

UT LISP was developed under a SCOPE-like operating system, and as much of its original code as possible has been retained in the 7600 implementation. Therefore, LISP is structured to run normally in batch mode, although excellent interactive capability has been incorporated. LISP uses CDC display code internally and converts to 6-bit ASCII on input and output. A code translation table has been included as an appendix to the manual.

MASTER

CHAPTER 2

USING THE LISP SYSTEM

(MODIFIED FOR USE AT LLL)

A copy of the binary code of LISP is stored on the photostore and may be accessed through ELF by the name:

.335821:LISP:LISP

2.1 THE LISP EXECUTE LINE

The LISP execute line has the form:

LISP P1 P2 P3 ... Pn / t v

The various p_i's are parameters controlling the mode of operation of LISP. Any number including zero, of the options listed below may appear in any order. If present, parameters must be separated by spaces. Each parameter has a default value which is assumed if it is omitted. Certain parameters require numeric values to be associated with them. In all such cases the number can be an unsigned decimal integer or an unsigned octal integer. Octal numbers may be written as a string of octal digits followed by a Q, B, or K. If the octal number is terminated by a K, it is multiplied by 1000₈ internally. Numbers are limited to a maximum of 10 characters. Some parameters are file names, or incorporate file names. All file names used within LISP are limited to 7 characters. The various parameters are listed below:

<u>Parameter</u>	<u>Form</u>	<u>Meaning</u>
Listing control	absent	no listing of source cards will appear
	L	listing of source lines will appear on SYSOUT
	P	same as L except a parenthesis level count will be printed under each parenthesis on the listing. This option is recommended for undebugged programs.

<u>Parameter</u>	<u>Form</u>	<u>Meaning</u>
Conversational mode control	absent	the action of LISP will be defined by the other parameters.
	C	the effect of this parameter is as though the following parameter sequence occurred: 1=TTY, 0=TTY. Normal input and output will be done via a teletype using a line length of 70 characters. This parameter is intended for use of LISP under the OCTOPUS system.
Input file control	absent	LISP normal input (SYSIN) on INPUT.
	I=filename	LISP normal input (SYSIN) on <u>filename</u> .
Output file control	absent	LISP normal output (SYSOUT) on file OUTPUT. Line length is 130 characters.
	O=filename	LISP normal output (SYSOUT) on <u>filename</u> . Line length is 70 characters.
Garbage collector message control	absent	no garbage collector messages will be printed.
	G	a message will be printed each time garbage collection occurs. Note that this parameter controls only the message. The garbage collector is always in operation.
Timing message control	absent	the action of LISP will be as defined by other parameters present.
	T	a timing message will be printed after each doublet is evaluated. It has the form *TIME: number where <u>number</u> is the time (microseconds) required for the last doublet.
allocation control	absent	94 percent of the total available space is allocated to the free storage list and 6 percent is

<u>Parameter</u>	<u>Form</u>	<u>Meaning</u>
		allocated to the full-word list. Available space is defined as the total memory size minus the size of the LISP interpreter minus the size of the stack minus initial binary program space.
	A=number	<u>number</u> percent of available space is allocated to the free storage list. The remainder of available space is allocated to the full-word list. (10 ≤ <u>number</u> ≤ 96).
Stack size specification	absent	the stack is 1000 ₁₀ words long.
	B=number	the stack is <u>number</u> words long.
Binary program space allocation	absent	no memory is allocated to binary program space initially.
	X=number	<u>number</u> words allocated to binary program space at initialization.
Subsystem load control	absent	no LISP subsystem is loaded.
	name	the LISP subsystem named <u>name</u> is loaded and the functions in it are made available to the user.
Expert mode control	absent	all LISP error detection facilities are operative.
	E	primitive operations on atoms are allowed. <u>Warning</u> : only the user who really knows what he is doing should ever use this parameter.
Top-level function selection	absent	top-level function is <u>eval</u> .
	Q	top-level function is <u>evalquote</u> .
Error fatality control	absent	LISP will try to recover and continue running after the occurrence of an error.
	F	the occurrence of any recoverable error at the top level will terminate the LISP run.

<u>Parameter</u>	<u>Form</u>	<u>Meaning</u>
Error interrupt control	absent	the action of LISP will be as defined by other parameters present.
	Z	if the C parameter is also present then a simulated interrupt will occur on the detection of any recoverable error.
Output suppression control	absent	the action of LISP will be as defined by other parameters present.
	N	only user-originated output and error messages will be directed to the system output file (SYSOUT).
Field length control	absent	LISP will have $\sim 1000_8$ words of working space.
	M=number	LISP will have <u>number</u> words of working space.

Field length is approximately given by the formula:

$$FLL = FLS = 32200_8 + \text{working space}$$

Examples:

1. LISP C / t v

This is the normal execute line for using LISP in its smallest size from a teletype.

2. LISP C G T M=30K / t v

This execute line will run LISP from a teletype with 30000_8 words of working space, timing messages, and messages from the garbage collector.

3. LISP P I=XYZ O=TTY T B=1500 M=100K / t v

This execute line causes LISP to read its source input from a disk file named XYZ, producing a listing with parenthesis counts on the teletypes. Timing messages will be generated, a 1500-word stack is used, and 10000_8 words of working space are allocated.

2.1.1 PARAMETER RESTRICTIONS

Some parameter combinations are not allowed and some other parameters may appear only once on the control card. Those that can appear only once each are: X, S, I, and O. Illegal combinations are C with either or both of I and O. If these restrictions are violated, LISP will abort with a message:

PARAMETER ERROR

The parameters L and P have no effect if they appear in combination with C or with N. Z has no effect unless it appears with C. The parameters A = number and B = number may appear more than once, and the last appearance determines their effect.

All other parameters may appear any number of times. If they appear more than once, their effect is the same as if they appeared only once.

2.1.2 NOTES ON LISP PARAMETERS

The allocation control parameter is used to vary the relative sizes of free space and full-word space in order to fit the working storage of LISP to the program at hand. For instance, a program which deals with many numbers will need relatively more full-word space than a program whose main purpose is the manipulation of lists. If one relies on the usual 6% of total available space allocated to full-word space, then the size of total memory necessary to get enough full-word space would be quite large and there would be much of the free space which would likely never be used. By decreasing the number associated with the A parameter on the control card, relatively more full-word space will be available to the program and it may be able to run in a smaller total memory than otherwise. To determine whether an adjustment of the allocation control parameter is advisable the user should consult the output from a previous run. LISP prints at the end of every run a summary of how many times the garbage collector was called when free space was exhausted and how many times it was called when full-word space was exhausted. If the number of times it was called due to full-word space exhaustion is significantly larger than the number of times it was called due to free space exhaustion, then the allocation control parameter should be decreased, otherwise the parameter should be increased.

The stack size specification parameter is used to vary the size of the stack in the LISP system. Increasing the stack increases the number of levels of recursion which are allowed in the LISP program. Remember that increasing the stack size will decrease the amount of free and full-word space available in the same field length.

The binary program space allocation parameter is used to allocate memory space for the storage of compiled LISP code (see Chapter 6). If no binary program space has been allocated, LISP will automatically request more memory from the operating system in which to store compiled code. If the user knows, however, approximately how many words of binary program he will be using, he can pre-allocate the space using this parameter and thereby save the overhead incurred by LISP requesting more space from the operating system.

The subsystem load control parameter offers mechanism for the user to initialize the LISP system to some set of functions and atom values other than the standard ones.

A LISP subsystem has a name which is the name of the file on which it resides. The subsystem consists of a series of doublets which are automatically read by LISP and executed before any of the user's doublets are executed.

A LISP subsystem can be loaded into any size of memory under any allocation parameter set that will accommodate it. The disadvantage of the subsystem is that it requires processing (i.e., extra time) by the LISP system each time it is loaded.

Expert mode control when present allows the user to perform car, cdr, csr, rplaca, rplacd, and rplacs operations on atoms. Normally, these operations are prohibited by LISP and will cause an ILLEGAL ARGUMENT error. Improper usage of these operations on literal atoms may cause changes in information vital to the operation of the LISP system, and may cause the program to fail catastrophically. Therefore, only the really expert user who has some valid reason for performing these operations should ever run with this parameter turned on.

The actions specified by several of the control card parameters may be turned on and off dynamically under user control during the course of a run. The listing control, garbage collector message control, timing message control, top-level function selection, and error interrupt control parameters may all be effected by functions described in Section 4.12. Input file control and output file control functions are described in Section 5.1.1.

2.2 INPUT FORMAT FOR UT LISP

UT LISP provides the user a choice of top-level supervisor functions, either eval or evalquote. The forms of input for these two cases are different.

eval is the usual top-level function and expects as its input a single LISP expression, or form, to evaluate. This form consists of a function name or lambda expression followed by a sequence of argument expressions and all enclosed in parentheses. For example:

```
(EQ (QUOTE A) (QUOTE B))
```

is a proper expression for eval. Note that constant items in the expression must be quoted to prevent their evaluation. Expressions will be processed in order one after another with each one being evaluated before the next is read. Only one expression is allowed on a line.

evalquote may be selected as the top-level function. It expects its input to be a so-called doublet which consists of a function name or lambda expression followed by a list of actual arguments for that function.

For example:

```
EQ (A B)
```

is the form used for the previous example under evalquote. No quoteing is required of the arguments. Doublets will be processed in order one after another with each being evaluated before the next is read. Only one doublet is allowed on a line.

Execution terminates when an end of file occurs on the normal input file (SYSIN) or when a line whose first 3 nonblank characters are FIN is read.

LISP functions and data may be placed in free format on input lines in columns 1-72. The only restriction is that an atom may not be split between two successive lines. In UT LISP, a literal atom is any string of thirty characters or less which contains no parentheses, periods, blanks, or commas and cannot be interpreted as a valid number. The syntax of the numeric atom is more or less the standard syntax of numbers. For further details on the syntax of literal atoms and numeric atoms the user is referred to Section 3.1. Figure 1 shows a representative input deck ready to be processed by LISP.

Any data that is to be read during evaluation of an expression or a doublet must immediately follow that expression or doublet on the normal input file unless it is to be read from a different file.

Certain characters have special meaning when encountered in LISP input. They are intended to make typing input data less tedious.

Character

Teletype

Meaning

"

the S-expression immediately following will be quoted. E.g., $\equiv(A)$ will be represented internally as (QUOTE (A)) and will print as (QUOTE (A)).

+

accept the immediately next character as part of a literal atom even if it is not normally allowed. E.g., A+.B will be the atom A.B.

[

equivalent to a left parenthesis, but marks a closure point for a following] character. E.g., [A] is equivalent to (A).

]

equivalent to an arbitrary number of right parentheses sufficient to match all left parentheses up to and including the most recent [character. If no [character precedes the], it will match all left parentheses remaining unmatched in the S-expression. E.g., [(((A] is equivalent to (((A)))

\\

while not a single character, two immediately adjacent \ characters have the meaning that the S-expression currently being input should be discarded. LISP will request input again. This is of primary benefit to the conversational user who may need to discard an erroneous S-expression which extends over several lines.

Also, if evalquote is being used as the top-level function the occurrence of the atom STOP as the second half of a doublet cancels the current evaluation. This is of greatest value to conversational users who need to recover from a typing error in the first half of the doublet.

2.3 OUTPUT TO BE EXPECTED FROM THE LISP SYSTEM

When LISP begins execution, it first sends a version message of the form:

UT LISP - VER. number (LLL)

to the system DAYFILE. Here number is the version number of the LISP system. These items allow the user to determine which version of LISP he is using.

As LISP executes each expression or doublet it produces output to inform the user of what it was that was requested and what the result of its evaluation was. At the beginning of each evaluation the system will print a line containing:

*EVAL: or *EVALQUOTE:

as appropriate for the top-level function in use. This line is a signal that the top-level function has begun operation. (In conversational mode this is a signal that LISP is ready for the user to type in new information. LISP types ? on the teletype as a prompt when it is ready for new input.) Following this line, the expression or doublet which is currently being evaluated will be printed. If a listing control parameter is present on the control card then printed here will be an exact image of what was read on the source lines. If listing control is not on then printed here will be the expression or doublet as printed by the normal LISP printing function. Expressions or doublets whose function is define will not be printed in this case as they are frequent and voluminous and the user should be acquainted with their content already. Next there will appear two blank lines followed by a line containing:

*VALUE:

This line signals the completion of evaluation. Following this line will be printed the result of the evaluation. After printing the result LISP, then skips two more blank lines, types the top-level message, and is ready for more input. If timing messages are requested via the T parameter on the execute line, they are printed between the two blank lines following the value.

At the end of the LISP run, the message:

GARBAGE COLLECTIONS: number₁ number₂

is printed. Number₁ is the number of times the garbage collector was activated due to free space exhaustion and number₂ is the number of times is was activated due to full-word space exhaustion.

Figure 2 shows the output obtained from the source in Figure 1.

CHAPTER 4

FUNCTION DEFINITIONS

All LISP programs are built by function composition, that is, the user defines functions which in turn call on other functions, either user-defined or defined in the LISP system. In this chapter are precisely described most of the functions defined in UT LISP. Some functions are described in Chapters 5, 6, and 7 as appropriate. These function descriptions should serve the user as his definitive guide to their use.

4.1 FUNCTION TYPES

There are four function types in UT LISP, distinguished by the names EXPR, SUBR, FEXPR, and FSUBR. Functions of type EXPR and FEXPR are stored in the machine as lambda expressions to be executed in an interpretive manner. Functions of type SUBR or FSUBR are stored in the system as machine code which is executed directly by the computer when requested. User functions are generally of type EXPR or FEXPR and the functions defined within the LISP system are always of type SUBR or FSUBR. Functions of type EXPR and SUBR are functions of a fixed number of arguments. Furthermore, at the time such a function is called, the argument expressions appearing in the calling expression will be evaluated before being given to the called function. Arguments of SUBR's may be omitted from the right, and the missing arguments will be supplied as NIL. Functions of type FEXPR and FSUBR on the other hand can have an arbitrary number of arguments in the calling expression and those arguments will not be evaluated before being given to the called function. FEXPR and FSUBR functions are also sometimes called "special forms".

LISP functions may further be classified as normal functions, pseudo-functions, or predicates. Normal functions receive some arguments, perform some manipulation on those arguments, and return a value which depends upon the manipulation performed. Pseudo-functions receive some arguments and perform some side-effect operation rather than some manipulation upon the arguments. Since all LISP functions must return some value, the pseudo-functions

will also return a value, but the value may bear no relationship to the arguments which were given it. Predicates are functions which receive some arguments and return as value either true or false depending on some relationship that holds among the arguments. In LISP, false is always represented by the atom NIL and truth is represented by anything other than NIL. Most predicates will in fact return the atom *T* as their value for truth, but the user should endeavor to not use this fact.

4.2 NOTATION TO BE USED IN THIS CHAPTER

Since the notation of LISP function usage involves two different forms, one to be used with eval and another to be used with evalquote, the function descriptions in this chapter will be given using a third notation with the understanding that the user must translate from this notation into either of the appropriate forms when he actually writes input for LISP. In this notation, function names will be given in lower case letters and will be followed by a description of the arguments expected by the function enclosed in square brackets. For example:

`cons[s1;s2]`

which indicates that cons is a function of two arguments, either of which may be any arbitrary S-expression. When appearing in the text of this manual LISP function names will be underlined. In describing the arguments expected by the function semicolons will be used to separate successive arguments and in the case of a function which can accept an arbitrary number of arguments the ellipsis (...) will be used to indicate this fact.

In general, a function will have certain restrictions as to what types of arguments it may receive. The descriptions given below will indicate these restrictions through a shorthand notation in the list of arguments. Where one of the symbols given in the table below appears in the list of arguments of a function, the argument actually supplied to the function must obey the characteristics defined for that symbol. If the restrictions are not met, the LISP system will probably detect an ILLEGAL ARGUMENT or ILL-FORMED ARGUMENT error.

TABLE OF ARGUMENT TYPES

<u>Type Code</u>	<u>Meaning</u>
atom	The argument must be an atom, either a literal atom or a numeric atom.
character	The argument must be a literal atom whose name is a single character.
exp	The argument must be some LISP expression which can be evaluated by <u>eval</u> .
flnumber	The argument must be a floating-point numeric atom.
fixnumber	The argument must be a fixed-point numeric atom (either integer or octal).
funcl	The argument must be a functional. It must be either the name of a function preceded by <u>quote</u> or <u>function</u> , or it must be a lambda expression preceded by <u>quote</u> , <u>fquote</u> , or <u>function</u> .
function	The argument must be a function name or a lambda expression.
fw	The argument must be a single full-word, that is a word in full-word space.
fwl	The argument must be a list of full-words.
lat	The argument must be a list of literal atoms.
letter	The argument must be a literal atom whose name is a single letter (A...Z).
list	The argument must be a list or NIL, the empty list. By list is meant a nonatomic S-expression whose right-most cdr is NIL, <u>i.e.</u> whose top-level printed representation contains no periods.
litatom	The argument must be a single literal atom.
nats	The argument must be some nonatomic S-expression.
number	The argument must be a numeric atom of any type.
pred	Same as exp, but the expression will be considered to return a truth value--NIL $\equiv$ false, anything non NIL $\equiv$ true.
s	The argument may be any arbitrary S-expression.

The function descriptions given in the remainder of this chapter will show the name of the function, the types of arguments it expects, the classifications of the function, and will give a precise description of the action of the function.

The function descriptions are grouped into subsections according to overall purpose. Within each group the descriptions are given of functions first, predicates last, and among these groups the most commonly-used appear first.

4.3 ELEMENTARY FUNCTIONS AND PREDICATES

The elementary functions and predicates comprise the LISP primitive functions and predicates on which the language is based and the most commonly-used nonprimitive functions and predicates.

`car[nats]`

Normal; SUBR

car returns as value the left part of the dotted pair which is equivalent to the argument. In list terms, it returns the first element of the list. It retrieves the value stored in the CAR field of the memory word which is its argument.

`cdr[nats]`

Normal; SUBR

cdr returns as its value the right part of the dotted pair which represents its argument. In list terms it returns the rest of the list after the first element is deleted. It functions by retrieving the value stored in the CDR field of the word which is its argument.

`csr[nats]`

Normal; SUBR

csr retrieves the value stored in the CSR field of the word which is its argument.

`caar[nats]`

`cdar[nats]`

.

.

.

`cdddddddr[nats]`

Normal; EXPR

Multiple car-cdr functions are allowed and may contain up to 8 a's, d's, or s's between the c and the r to denote a sequence of car, cdr, or csr operations. The sequence of operations is applied in right to left order that is: `cdar[nats] = cdr[car[nats]]`.

cons[s₁;s₂]

Normal; SUBR

cons builds a new S-expression. It constructs the dotted pair (s₁ . s₂) by obtaining a new word from free space and storing its first argument into the CAR field of the word and its second argument into the CDR field of the word. This is the fundamental function for building new S-expressions.

list[s₁;s₂;...s_n]

Normal; FSUBR

list takes an arbitrary number of arguments and constructs a new list such that s₁ is the first element, s₂ is the second element, and so on. If no arguments are given the value is NIL.

quote [s]

Normal; FSUBR

quote is essentially a do-nothing. It receives its argument unchanged and returns that same argument unchanged as its value. quote is used to prevent evaluation of S-expressions which serve as data internal to a LISP function definition. Within a LISP function, most literal atoms will be evaluated as variables and lists will be evaluated as function calls. When quote precedes an atom or an S-expression it is a signal to the interpreter that these things are not to be evaluated as variables and function calls, but instead are actual constant data of the function.

fquote[s]

Normal; FSUBR

fquote is identical to quote with the exception that the compiler will compile the argument of fquote as a function when compilation is occurring and will not compile the argument of quote. Therefore, in functions which are to be compiled, fquote should appear only before lambda expressions.

set[atom;exp]
cset[atom;exp]

Pseudofunction; SUBR

set is the principal value-assignment function. Its first argument must evaluate to an atom and the value of the second argument is made the new value of the atom. The value of set is its second argument, or the new value which was assigned. cset is a synonym for set.

setq[atom;exp]
csetq[atom;exp]

Pseudofunction; FSUBR

setq is similar to set except that its first argument is quoted. That is, the first argument is the actual atom which will have its value assigned. The value returned by setq is its second argument, or the new value which was assigned to the variable. csetq is a synonym for setq.

$setq[X;Y] \equiv set[quote[X]; Y]$

rplaca[nats;s]

Normal; SUBR

rplaca replaces the CAR field of its first argument with a pointer to its second argument. This function modifies existing list structure in memory, and the user should take great care in using it. The value returned is the first argument of rplaca but this argument has been changed by the action of the function.

rplacd[nats;s]

Normal; SUBR

rplacd replaces the CDR field of its first argument with a pointer to its second argument. This function modifies existing list structure in memory. rplacd should be used very carefully by the user as it may be used to create circular lists, which may cause difficulty with other processes. The value returned by rplacd is its first argument which has been modified by the action of the function.

`rplacs[nats;s]`

Normal; SUBR

rplacs replaces the CSR field of its first argument with a pointer to the second argument. This function is the only function which allows the user to place information into the CSR field. It modifies existing list structure in memory and therefore should be used with caution. The value returned by rplacs is its first argument which has been modified by the action of the function.

`atom[s]`

Predicate; SUBR

atom returns true if its argument is either a literal atom or a numeric atom. It returns false if its argument is any other expression.

`eq[s1;s2]`

Predicate; SUBR

eq returns true if its two arguments are the same literal atom. It returns false under any other circumstance. Literal atoms are stored uniquely in the LISP system so that their equality may be determined by simple comparison of machine addresses. No other type of S-expression is stored uniquely and therefore will usually fail to give true under eq comparison.

`eqn[s1;s2]`

Predicate; SUBR

eqn is identical to eq except that it also works for numeric atoms. It will return true either if its two arguments are the same literal atom or if they are two numeric atoms which have the same numeric value. The two numbers may be of different types. eqn will return false under all other circumstances.

`equal[s1;s2]`

Predicate; SUBR

equal returns true if its two arguments are equivalent S-expressions.

Two S-expressions are equivalent if they are both the same literal atom, if they are two numbers with the same value, or if they are two non-atomic S-expressions composed of the same atoms in corresponding positions.

equal returns false if its two arguments are not equivalent S-expressions.

`null[s]`

Predicate; SUBR

null returns true if its argument is the atom NIL. It returns false under all other circumstances. It is a logical negation operator.

`numberp[s]`

Predicate; SUBR

numberp returns true if its argument is a number of any type. It returns false if its argument is any other kind of S-expression.

`member[s;list]`

Predicate; SUBR

member searches the list for an occurrence of the S-expression on its top level. If the S-expression is not there the value of member is NIL. If it is present, the value of member is the remaining portion of the list beginning with the element sought.

`alphap[litatom1; litatom2]`

Predicate; SUBR

alphap compares the print names of its two arguments and returns true if the first argument alphabetically precedes the second argument. It returns false if the first argument alphabetically follows the second or if they are the same atom. The collating sequence of the CDC 6000 series display code is used to determine the alphabetization.

`gradp[s1;s2]`

Predicate; SUBR

gradp returns true if its first argument resides at a lower memory address than its second argument. It returns false if its first argument resides at the same or higher memory address than the second argument. This function can be used as an arbitrary but consistent ordering predicate since a given S-expression will always reside at the same address during the course of one LISP run.

4.4 LOGICAL CONNECTIVE FUNCTIONS

The logical connective functions are used to form complex Boolean expressions from simple predicate expressions.

not[s]

Normal; SUBR

not is in every way equivalent to the predicate null. It is the logical inversion operator. It returns true if its argument is false and returns false if its argument is true.

and[exp₁;exp₂;...;exp_n]

Normal; FSUBR

and evaluates each of its arguments in turn until it encounters the first argument which is false or until it evaluates all of its arguments. If all of its arguments evaluate to true the value of and is the value of exp_n (which is true). If one argument is false the value of and is false. Any expressions following the first expression which gives a false value will not be evaluated. If there are no arguments the value of and is true.

or[exp₁;exp₂;...;exp_n]

Normal; FSUBR

or evaluates each of its arguments in turn until it finds the first argument which has a true value. If one of its arguments is true, then the value of or is the value of that argument. If none of its arguments is true, then the value of or is false. Any arguments following the first argument which is true will not be evaluated. If no arguments are present the value of or is false.

4.5 SEQUENCE CONTROL AND FUNCTION EVALUATION

The functions of this section are used to control the sequence of execution of expressions in a LISP function by providing conditional and iterative control structures. Also included are functions which enable the user to construct and evaluate his own functions and expressions at run-time.

`cond[(pred1 exp...exp) (pred2 exp...exp)...(predn exp...exp)]`

Normal; FSUBR

cond is one of the major functions of LISP. Most user-defined functions are defined in terms of a cond-expression (conditional expression). cond provides a conditional control structure similar to that provided by the IF statement of Algol. The arguments of cond are lists of expressions containing at least one expression each. cond proceeds as follows: pred₁ is evaluated and if it is true then the remaining expressions (if any) in the list containing pred₁ are evaluated in left to right order. If pred₁ is false then pred₂ is evaluated and if it is true the expressions following it (if any) are evaluated. This process continues for pred₃, pred₄, etc., until either some pred_i is true or all pred_i's have been evaluated as false. In any case the value of cond will always be the value of the last expression evaluated under its control. The value of cond with no arguments given is NIL.

`select[exp; (exp1 exp2); (exp3 exp4); ...; (expn expn+1); expn+2]`

Normal; FSUBR

select allows the selection of a particular expression to be evaluated depending on the value of the first argument of select. It proceeds as follows. The first argument is evaluated. Its value is then compared to the value of exp₁, exp₃, ... until it is found to be equal to one of the exp_i's in which case exp_{i+1} is evaluated and returned as the value of select. If the value of exp is not equal to the values of any of the exp_i's then exp_{n+2} is evaluated and returned as the result of select.

prog2[exp₁; exp₂]

Pseudo-function; SUBR

Both arguments of prog2 are evaluated and the value of the second one is returned as the value of prog2. This function is used to perform two actions where LISP normally allows only one.

progn[exp₁;exp₂;...;exp_n]

Pseudo-function; FSUBR

progn evaluates each of its arguments in turn and the value of the last argument is returned as the value of progn. This function is used to perform a simple sequence of actions.

prog[lat;s₁;s₂;...;s_n]

Pseudo-function; FSUBR

The prog expression is used to form iterative functions. The first argument is a list of atoms which serve as temporary variables within the prog expression. The remaining arguments are either literal atoms or else they are expressions to be evaluated. Literal atoms serve as labels and the expressions serve as statements of a sequential programming language. prog proceeds by first assigning the value NIL to each of the temporary variables. It then evaluates each of the s_i's unless they are atoms. Atoms are skipped over and the expressions following are evaluated, one after the other. The use of the go function allows the sequence of execution to be altered (see below). The value of a prog expression will be NIL if all of the s_i's are executed and none are left unless one of them happens to be a call on the return function in which case the argument of return will be the value of the entire prog expression. go and return may be executed in other functions which are called by the function containing a prog expression.

go[litatom]

Pseudo-function; FSUBR

go is used to control execution sequence within a prog expression. Its argument is not evaluated and must be one of the labels which appears in the prog expression. Execution of go causes the execution of the prog expression to be continued with the statement immediately following the atom which is the argument of the go. go has no value as such. go may not be used to branch out of a prog expression, but may transfer control only within the most recently occurring prog expression.

return[exp]

Pseudo-function; SUBR

return causes LISP to exit a prog expression. The argument of return is made the value of the entire prog expression. return may be executed within a non-prog expression which is evaluated under the control of a prog expression in which case LISP exits from the prog expression.

eval[exp]

Normal; SUBR

eval evaluates its argument which must be a LISP-evaluable expression and returns the value of the expression as its value. Variables within the expression will be evaluated in the context in which the call to eval is made.

apply[function; list]

Normal; SUBR

The second argument serves as a list of arguments for the function which is the first argument of apply. The value of apply is the value obtained by applying the first argument to the second. The first argument must be either a function name or a lambda expression and if it is a function name the function must be of type FEXPR or SUBR. If a function of type FEXPR or FSUBR is given to apply then an UNDEFINED FUNCTION error will occur even though the function may in fact be defined.

evalquote[function; list]

Normal; SUBR

evalquote is similar to apply in that it applies the function given as its first argument to the list given as its second argument. evalquote however, allows the function to be of type FEXPR or FSUBR as well as EXPR or SUBR. The result of the application of the first argument to the second argument is the value of evalquote.

evlis[list]

Normal; SUBR

evlis expects its argument to be a list of expressions. It evaluates these expressions one at a time from left to right and returns a list whose elements are the respective values of the expressions.

function[function]

Pseudo-function; FSUBR

The argument of function is expected to be a function name or a lambda expression. This function is required in certain very special cases to prepare an environment for a functional argument being passed to another function. Its value is a so-called FUNARG expression.

4.6 LIST MANIPULATION FUNCTIONS

List manipulation functions are the principal means for dealing with S-expressions. Operations of construction, copying, reversal, and combination are possible.

`length[s]`

Normal; SUBR

The argument of length may be either an atom or a list. If a list, then length returns as its value the number of elements in the list. If the argument is an atom, the value of length is zero.

`copy[s]`

Normal; SUBR

copy copies its argument. That is, it produces an entirely new list structure occupying different memory words which is equivalent to the original s-expression.

`reverse[list]`

Normal; SUBR

reverse produces a new list whose top level elements are the same as the elements of its argument except appearing in reverse order. Sublists of the list are not themselves reversed.

`append[list; s]`

Normal; SUBR

The first argument of append must be a list. The second argument normally is a list, but does not have to be. append joins its two arguments together such that the resulting expression contains the elements of both arguments with the elements of the first argument appearing before the elements of the second argument. The first argument is copied and its terminal NIL is replaced by a pointer to the second argument.

`nconc[list; s]`

Normal; SUBR

nconc is similar to append in that it joins its two arguments into a single S-expression. nconc actually modifies the first argument to replace the terminal NIL with a pointer to the second argument. The value of nconc is its first argument, but the first argument has been modified by the action of the function. This can be contrasted with the value of append which creates a new list structure and does not modify either of its arguments.

`conc[list1; list2; ...; listn]`

Normal; FSUBR

conc is similar to nconc in that it concatenates its arguments together except that it can accept an arbitrary number of lists and produces a result which is the concatenation of all of the lists in the order that they are presented. conc modifies the existing list structure.

`pair[list1; list2]`

Normal; SUBR

pair matches its two argument lists together producing a result list whose length is equal to the length of its shortest argument. Each element of the resulting list will be a dotted pair whose car is the corresponding element of the first argument list and whose cdr is the element of the second argument list. Thus, the first elements are paired together and the second elements are paired together and so forth.

`efface[s; list]`

Normal; SUBR

efface removes the first occurrence of the item `s` from the list. efface modifies the existing structure of the list.

sublis[((s₁ . s₂)(s₃ . s₄) ... (s_{n-1} . s_n)); s_{n+1}]

Normal; SUBR

sublis receives a list of dotted pairs, as indicated, for its first argument and an arbitrary S-expression as its second argument. The value of sublis is the result of substituting the right part of a dotted pair for every occurrence of the left part of the dotted pair in the second argument (s_{n+1}) of sublis. Thus, sublis allows for the simultaneous substitution of a number of items in an arbitrary S-expression. sublis does not modify existing memory structure, but creates a new structure containing the various substitutions.

sublis[((A . 1) (B . 2) (C . 3)); (A (B C B) A)] = (1 (2 3 2) 1)

subst[s₁;s₂;s₃]

Normal; SUBR

The value of subst is the result of substituting s₁ for all occurrences of s₂ in s₃. subst does not modify existing list structure but creates an entirely new list structure containing the substitutions.

subst[A; B; (A (B C B) A)] = (A (A C A) A)

4.7 PROPERTY LIST MANIPULATION FUNCTIONS

Every literal atom in the system has a property list associated with it. The property list contains items of information about, or properties of, the atom. Users may assign properties of their own choosing to atoms. Each property is characterized by an indicator which is itself a literal atom and defines what property is being mentioned, and a value for that property which may be any S-expression. The indicators INFO, PNAME, EXPR, SUBR, FEXPR, FSUBR, SYM, CMACRO, SMACRO, CSUBR, and CFSUBR have special meaning to the LISP system itself and should be avoided or used with care. Property lists do not have the same structure (see Chapter 3) as ordinary lists and should be manipulated only by the following functions.

`put[litatom1;litatom2;s]`

pseudofunction; SUBR

put places a new element on the property list of litatom₁. It uses litatom₂ as the property indicator and the third argument, an S-expression, is made the property value. put deletes any other property list element with the same indicator. The value of put is its first argument.

`get[litatom1; litatom2]`

pseudofunction; SUBR

get is the inverse of put. get searches the property list of litatom₁ for a property list element whose indicator is litatom₂. The value of get is the value associated with that property indicator if it is present. Otherwise the value of get is NIL. get should not be used to obtain the PNAME property of an atom if the PNAME is going to be used with the character manipulation functions (see section 4.10). Instead the function getpn (see below) should be used.

```
deflist[(((litatom1 s1) (litatom2 s2)... (litatomn sn)); litatomn+1]  
pseudofunction; SUBR
```

deflist is used to simultaneously assign a new property to a number of different atoms. The first argument is a list of sublists, each of two elements as shown above. The second element of each of the sublists is placed on the property list of the first element of the sublist as the value of a property whose indicator is the second argument of deflist, that is, litatom_{n+1}. The value of deflist is a list of the litatom_i's which appear in the first argument, that is, a list of all of the atoms to which the new property has been assigned by deflist.

```
define[(((litatom1 function1)(litatom2 function2)... (litatomn functionn)))]  
pseudofunction; SUBR
```

The first argument of define is equivalent in form to that of deflist with the exception that the second element of each of the sublists must be a function, that is either the name of a function or a lambda expression. define makes function_i a property of litatom_i with the indicator EXPR. Its primary purpose is the definition of LISP functions. define is equivalent to a call on the function deflist with the second argument being the atom EXPR. The value of define is a list of the atoms which have received the function definitions.

```
getpn[litatom]
```

pseudofunction; SUBR

getpn returns the PNAME property of the litatom structured as a list of full words, that is a list whose CAR fields point directly to the full words which contains the characters of the print name of the litatom. Such a list structure is necessary for manipulation by some of the functions defined in section 4.10.

remprop[litatom₁; litatom₂]

pseudofunction; SUBR

remprop removes the property whose indicator is litatom₂ from litatom₁.
The value of remprop is always NIL.

flag[lat; litatom]

pseudofunction; SUBR

flag puts its second argument on the property list of each of the atoms in its first argument as a property indicator whose value is *T*. Thus, a flag is a marker property which is either there or not there, but which has no particular value. No flag will be duplicated on the property list. The value of the function flag is NIL.

remflag[lat; litatom]

pseudofunction; SUBR

remflag removes all occurrences of the indicator litatom from the property list of each of the atoms in the lat. The value of remflag is always NIL.

prop[litatom₁;litatom₂;func1]

normal; SUBR

prop is similar to get in action. The property list of litatom₁, is searched for the indicator litatom₂. If such a property is found, the entire property list of litatom₁, beginning with property litatom₂ is returned by prop. If litatom₂ is not an indicator on the property list of litatom₁, then func1 which must be a function of no arguments is applied and its value returned.

4.8 FUNCTIONS WITH FUNCTIONAL ARGUMENTS

These functions all sequentially scan a list performing an operation on the list with each scanning operation. The operation to be performed is specified by a functional argument. Thus these functions have variable semantics. Although function must be used with functional arguments for some user-defined functions (see Appendix D), it is never necessary with these functions.

map[list; func1]

pseudofunction; SUBR

map applies the functional which is its second argument to the list which is its first argument. Then, it applies the functional to cdr of its first argument, then to cddr, etc. until the functional has been applied to all possible cdr's of the list. The value of map is always NIL and it is always used to perform some kind of side effect computation. The second argument must be a function of only one argument.

maplist[list; func1]

normal; SUBR

maplist is similar to map except that its value is a list of each of the values produced when the second argument is applied to successive cdr's of the list. These elements are cons'ed together as they are generated. The second argument must be a function of only one argument.

mapcon[list; func1]

normal; SUBR

mapcon is similar to maplist except that the value is the list which results from concatenating the individual results of applying the function to successive cdr's of the list. This implies that each value returned by the functional must itself be a list or else erroneous results may occur when the elements are concatenated together. The second argument must be a function of one argument.

mapcar[list; func1]

normal; SUBR

mapcar is similar to maplist in that it returns a list of the results of the individual applications of the functional. However, the function is applied at each step to car of the successive cdr's of the list instead of to the cdr itself. The second argument of mapcar must be a function of only one argument.

search[list; func1₁; func1₂; func1₃]

normal; SUBR

search applies func1₁ to successive cdr's of the list until it happens that the value of func1₁ is true. If this condition occurs then search applies func1₂ to the remainder of the list at that time. If the entire list is exhausted without any application of func1₁ giving a true value then func1₃ is applied to the list at that point which, of course, is always NIL. search is therefore used to apply some function to a portion of the list depending on a condition which must be met by some elements of the list. All of the functionals must be functions of one argument.

sassoc[s₁; ((s . s) (s . s)...(s . s)); func1]

normal; SUBR

sassoc searches its second argument (which as shown is a list of dotted pairs) for a pair whose first element is equal to s₁. If such a pair is found the value of sassoc is that pair. Otherwise, the function func1 of no arguments is applied and its value returned as the value of sassoc.

4.9 ARITHMETIC FUNCTIONS AND PREDICATES

All of the functions described in this section must receive numeric atoms as their arguments. There are three types of numbers in LISP: octal numbers, integers, and floating-point numbers. Octal numbers and integers are not distinguished for arithmetic operations; both are considered to be fixed-point. They differ only in their printed representation. All of the arithmetic functions may accept either fixed-point or floating-point inputs or a mixture of the two. The result will be floating-point if any of the inputs are floating-point and will be fixed-point otherwise.

`add1[number]`

normal; SUBR

add1 increments its argument by 1 and returns the incremented value as its result. A fixed-point result will be of integer type.

`sub1[number]`

normal; SUBR

sub1 decrements its argument by 1 and returns the decremented quantity as its value. A fixed-point result will be of integer type.

`plus[number; number; ...; number]`

normal; FSUBR

plus computes the sum of all of its arguments no matter how many there are and returns the sum as its value. A fixed-point result will be of integer type.

`difference[number; number]`

normal; SUBR

difference computes the difference of its two numeric arguments. (first minus second). A fixed-point result will be of integer type.

minus[number]

normal; SUBR

minus negates its argument. That is, it returns the negative of its argument. A fixed-point result will be of integer type.

times[number;number;...;number]

normal; FSUBR

times computes the product of all of its arguments. A fixed-point result will be of integer type.

quotient[number;number]

normal; SUBR

quotient computes the quotient of its two arguments. (The first divided by the second.) The quotient of two fixed-point numbers is a number of integer type which is the integer part of the actual quotient. Thus, quotient[1; 2] will return the value 0.

remainder[number;number]

normal; SUBR

remainder computes the remainder of the division of its first argument by its second argument. The remainder is calculated by the usual formula:

$$\text{remainder} = \text{dividend} - \lfloor \text{quotient} \rfloor \times \text{divisor}$$

A fixed-point result will be of integer type.

divide[number;number]

normal; SUBR

divide returns a list of two numbers wherein the first element is the quotient of the two arguments of divide and the second element is the remainder of the division.

recip[number]

normal; SUBR

recip calculates the reciprocal of its argument. That is, it is equivalent to quotient [1;number]. The reciprocal of any fixed-point quantity larger than one will be an integer zero.

max[number;number;...;number]

normal; FSUBR

max returns as its value the algebraically largest number in the set of numbers given to it.

min[number;number;...;number]

normal; FSUBR

min returns as its value the algebraically smallest number of the set of numbers given to it.

logand[number;number;...;number]

normal; FSUBR

logand computes the bit-by-bit logical product (logical and) of all its arguments from left to right. Each number is treated as a 60-bit quantity. The value of logand is always of octal type.

logor[number;number;...;number]

normal; FSUBR

logor computes the bit-by-bit logical sum (inclusive or) of its arguments in left to right order. Each number is treated as a 60-bit quantity. The value of logor is always of octal type.

logxor[number;number;...;number]

normal; FSUBR

logxor computes the bit-by-bit logical difference (exclusive or) of its arguments in left to right order with association to the left. Each number is treated as a 60-bit quantity. The value of logxor is always of octal type.

`leftshift[number; fixnumber]`

normal; SUBR

leftshift performs a shifting operation on its first argument. The second argument of the function is a shift count. If the shift count is positive the shift will be a left shift end-around circular. If negative the shift will be a right shift end-off with sign extension. The result of the leftshift function is always of octal type.

`fix[flnumber]`

normal; SUBR

The value of fix is the largest integer contained in the floating point number. The value is of integer type.

`float[fixnumber]`

normal; SUBR

float returns a floating-point number whose value is the same as that of the fixed-point argument. The fixed-point number must be less than 2^{48} in magnitude for this function to give the proper result.

`octal[number]`

pseudofunction; SUBR

octal converts its argument into a number of octal type. The print image of the resulting number is the octal representation of the original value. No conversion from floating-point to fixed point is made. octal actually modifies its argument directly. It does not make a copy of the argument.

`random[number]`

normal; SUBR

random returns a new random number in the range 0-1 each time it is called with a zero argument. If called with an argument between 0 and 1 it will return that argument as its value and on subsequent calls with a 0 argument it will return new random numbers belonging to a new sequence begun by the call with a nonzero argument.

zerop[number]

predicate; SUBR

zerop returns true as its value if its argument is a numeric zero. It returns false otherwise.

onep[number]

predicate; SUBR

onep returns true if its argument has the value 1 (either fixed-point or floating-point). It returns false if its argument is either greater than or less than 1.

minusp[number]

predicate; SUBR

minusp gives the value true if its argument is a negative number. It returns false if its argument is a positive number. Only the sign of the number is tested. Therefore, a -0 will cause minusp to return true.

fixp[number]

predicate; SUBR

fixp returns true if its argument is a fixed-point number (either an integer or an octal number). It returns false otherwise.

floatp[number]

predicate; SUBR

floatp returns true if its argument is a floating-point number and false otherwise.

lessp[number; number]

lessp returns true if its first argument is strictly less than its second argument. If either of the arguments is floating-point the comparison will be done using floating-point arithmetic.

greaterp[number; number]

predicate; SUBR

greaterp returns true if its first argument is strictly greater than its second argument. If either of the arguments is a floating-point number the comparison will be done using floating-point arithmetic.

4.10 CHARACTER MANIPULATION FUNCTION

Normally in LISP one is not concerned with the constituents of the atomic symbols. They are treated as indivisible units. On occasion, however, there does arise a need for decomposing an atom into its constituent characters or for taking a group of individual characters and building them into atomic symbols. The functions described in this section assist in these kinds of operations. A character in LISP is represented by a literal atom whose print name is the single character. Thus, the characters may be manipulated by any of the functions which may be used to manipulate atomic symbols. This means also that there is a difference between single digit numbers and the literal atoms whose names are the digit characters. The user should keep this difference in mind. There exists an internal buffer into which up to 120 characters may be placed. This buffer is useful for accumulating characters which are being used to construct atoms, and this buffer is manipulated by some of the functions described in this section.

`clearbuff[ ]`

pseudofunction; SUBR

This function of no argument clears the internal buffer and resets the counter of its contents to zero. The value of clearbuff is NIL. This function should be executed before any of the other functions which affect the buffer are used.

`pack[character]`

pseudofunction; SUBR

pack places the character which is its argument into the next sequential position in the internal buffer following all characters which have previously been placed there. The value of pack is NIL.

`mlenam[ ]`

pseudofunction; SUBR

This function takes the contents of the internal buffer and creates a list of full-words. The characters are taken from the buffer ten at a

time and placed into full-words containing up to ten display character codes. The last full-word is filled out with zeros if necessary. The resulting list is one whose CAR fields point directly to the full-words containing the character codes. This type of list structure is not a normal LISP list. The internal buffer is cleared after this function is completed.

intern[fwl]

pseudofunction; SUBR

The argument to intern must be a full-word list as returned by mknam (or getpn). intern takes this full-word list and produces a literal atom whose print name is given by the full-word list. intern places the atom on the OBLIST if necessary or returns a pointer to a pre-existing atom which has that print name.

numob[]

pseudofunction; SUBR

numob expects the internal buffer to contain a sequence of characters defining a number. The value of this function is the LISP number which corresponds to that character representation. The syntax of the numbers is the same as if the characters had been punched on a data card. If the characters in the buffer do not form a legal LISP number, then numob will return a literal atom whose print name contains those characters. The buffer is empty after this function is complete.

unpack[fw]

pseudofunction; SUBR

The argument to unpack must itself be a full-word, that is a memory word containing up to ten display code characters, filled with zeros if less than ten characters are present. The value of unpack is a list of literal atoms, one atom for each character contained in the argument of unpack. The print names of the literal atoms in the list correspond to the character codes in the full-word.

image1[atom;s]

pseudofunction; SUBR

image1 returns the integer length of the printed representation of the atom which is its first argument. If the second argument is NIL the image used will be the normal printed image. If the second argument is not NIL the image used will include any delimiter or escape symbols necessary to reproduce a readable form of the atom. This function is most useful in aiding the formatting of printed output.

numtoatom[number]

pseudofunction; SUBR

numtoatom creates a literal atom whose print name is equivalent to the representation number would have if printed. The format used for floating-point numbers is under the control of nformat (sec. 5.8).

liter[s]

predicate; SUBR

liter returns a value of true if the argument is a literal atom whose name is a single character which is a letter (A-Z). The value is false in all other circumstances.

digit[s]

predicate; SUBR

digit returns true if its argument is a literal atom whose print name is a single character which is one of the decimal digits (0-9). The value of digit is false under all other circumstances. Note that digit is false for the single-digit numbers 0-9.

opchar[s]

predicate; SUBR

opchar returns true if its argument is a literal atom whose print name is a single character and is either +, -, /, or *. opchar returns false under all other circumstances.

4.11 SYSTEM CONTROL, DEBUGGING, AND ERROR PROCESSING FUNCTIONS

These functions are used to give the user some control over the operation of the LISP system or to enable him to determine more information about the operation of his functions.

error[s]

pseudofunction; SUBR

error causes a recoverable error to occur. The argument to error will appear as part of an error message, printed in response to this function. Continued execution of the current doublet will be suspended unless the error function is executed under the control of an errorset (see below). This function has no value as such.

errorset[exp; s₁; s₂]

pseudofunction; SUBR

errorset is a function which allows the LISP system to recover from a recoverable error without terminating the execution of the current doublet. The first argument of errorset is evaluated. If no error occurs during the evaluation, then the result of errorset will be a list of the value of the expression. If an error does occur during the evaluation the result of errorset will be NIL. Any computation which causes an error to occur will cause the system to return to the point at which it entered the last errorset function. The second argument controls the printing of the error message in response to the error detected. If the second argument is true the error diagnostic message will be printed; if false, the error message will not be printed. The third argument controls printing of the backtrace with the error message. If both s₁ and s₂ are true, the backtrace will also be printed. By using errorset it is possible to continue computation after an error occurs. Note that errorset is a SUBR. Therefore, its first argument which is the expression to be evaluated

must be the result of evaluating some other expression in the calling expression for errorset. Thus, if the expression to be evaluated is desired to be passed directly to errorset it must be quoted in the calling expression.

trace[lat]

pseudofunction; SUBR

trace flags the property list of each of the atoms in its argument with the indicator TRACE. Subsequently, for those functions of type EXPR and SUBR whose names were contained in the argument of trace, the actual arguments supplied to that function will be printed every time that function is entered and the actual value resulting from that function will be printed every time the function is exited. This enables the user to get a complete history of the activity of a selected set of functions. The format of the printing is as follows:

[xx] ARGUMENTS OF name

This line is printed each time a traced function is entered. name represents the name of the function. xx is a two-character code which is changed every time a new "ARGUMENTS OF" message appears and it is designed to enable the user to tell which occurrence of a call on the function is being described. The xx begins with ! A and sequences through ! Z, etc. up to ;;. Following this line will be printed each of the arguments of the indicated function as actually received by that function. Each argument will be printed on a separate line.

[XX] VALUE OF name

This line is printed each time a traced function is exited. name represents the name of the function which is being exited. The actual value returned by the function at this point will be printed

beginning on the next line. These two messages appear in pairs. Every function that is entered must ultimately be exited. The indicator `xx` is used to match the corresponding lines together as they may occur separated by many pages of output if extensive tracing is used. Although tracing of functions can be very useful in the debugging process, the user is cautioned to be careful in his use of tracing as the extra printing consumes both time and paper. If too much tracing is done the user may wind up with a large amount of output which has too much detail. To be effective, the `TRACE` indicator must be placed on the property list of functions after those functions have been defined. The value of trace is always `NIL`.

`untrace[lat]`

pseudofunction; SUBR

untrace is the inverse of `trace`. It removes the flag `TRACE` from the property list of each atom in its argument. Any of those atoms which are function names will subsequently not be traced when applied as a function. The value of untrace is always `NIL`.

`traceset[lat]`

pseudofunction; SUBR

The atoms in the argument of traceset should be the names of functions which are composed of a prog expression. These functions are marked in such a way that while they are in control, every execution of a set or setq will cause a message to be printed, showing what variable was set to what value at that point in time. The message consists of the variable name followed by an equal sign followed by the value which was assigned to that variable. This process is useful for tracing the internal operation of functions written using prog expressions. The value of traceset is its argument unchanged.

untraceset[lat]

pseudofunction; SUBR

untraceset is the reverse of traceset. The atoms in its argument should be the names of functions composed of prog expressions. untraceset removes the marks set by the traceset function so that subsequent execution of those functions will no longer cause the printing that was done in response to the traceset. The value of untraceset is its argument unchanged.

4.12 MISCELLANEOUS FUNCTIONS

These functions provide the user with various capabilities for affecting or gaining information about system operation.

tempus[]

pseudofunction; SUBR

This function of no argument returns an integer value which is the CPU time in microseconds since the LISP run was begun.

tim[fixnumber]

pseudofunction; SUBR

tim allows the user to obtain various job timing information from the operating system. The argument must be in the range 0-4. The values returned by tim are given in the table below:

<u>Argument</u>	<u>Value</u>	<u>Type of value</u>
0	CPU time used by job (microseconds)	integer
1	I/O time used by job (microseconds)	integer
2	System time used by job (microseconds)	integer
3	Clock time (HH:MM:SS)	literal atom
4	Machine-date designator (LMM/DD/YY)	literal atom

reclaim[]

pseudofunction; SUBR

This function of no argument causes a garbage collection to occur whether it is needed or not. The value of reclaim is always NIL.

If the garbage collector message control is in effect, then a line will be printed showing the number of words collected by the garbage collector.

full[]

pseudofunction; SUBR

This function of no argument returns an integer result which is the number of full words currently available. This function consumes one full word and one free word each time it is called.

free[]

pseudofunction; SUBR

This function of no arguments returns an integer value which is the number of free words currently available. This function consumes one free word and one full word each time it is called.

remob[litatom]

pseudofunction; SUBR

This function removes its argument from the OBLIST. The net effect is that if the atom is not a member of any other list structure in memory then the words comprising that atom will be collected by the garbage collector upon the next garbage collection. Furthermore, if subsequently an atom with the same name is read an entirely new atom will be created and the old one will not be assumed. The value of remob is NIL.

addr[s]

pseudofunction; SUBR

This function returns an octal number as its value which is a representation of the address of the S-expression which is its argument. It enables one to determine the actual machine address of any given S-expression.

gensym[letter]

pseudofunction; FSUBR

gensym creates an entirely new literal atom each time it is called. The literal atom has a name of the form xnnnnnn where x is a single letter and nnnnnn is an integer. The letter used to compose the name is the argument of gensym. Each time gensym is called with no argument or with some letter as an argument, it generates the next atom in the series for that particular letter or for the last letter that was specified. Each atom in the series differs from the previous one by having the integer part of the name being incremented by one. If no argument is ever specified for gensym the letter G will be used. The atoms created by gensym are not placed on the OBLIST. Therefore, if an atom with the same name is subsequently read in, it will not correspond to the atom created by gensym.

alist[]

psuedofunction; SUBR

This function of no argument creates an association list which shows all variable bindings at the time the function is called. The function proceeds by scanning the OBLIST and creating a dotted pair for every atom which currently has a binding, each consisting of the atom dotted with its binding, thus, a call on this function will return a dotted pair list reflecting the current evaluation environment of the LISP system.

toplevel[s]

pseudofunction; SUBR

This function of one argument switches the top-level function of the LISP system between eval and evalquote. If the argument is NIL, the top-level function will subsequently be eval, otherwise it will be evalquote. The value of toplevel is always NIL.

saving[s]

pseudofunction; SUBR

If the argument of saving is not NIL, then the system will subsequently automatically bind the result of each top level evaluation to the atom PREVIOUS. This facilitates use of those results in the subsequent evaluation. If the argument is NIL this mode of operation will be discontinued.

mainloop[]

pseudofunction; SUBR

This function is the actual main loop of the LISP system. It is LISP-callable and has value NIL. While executing it expects to read and evaluate expressions or doublets. mainloop remains in control until it reads FIN or an end of file from SYSIN.

timing[s]

pseudofunction; SUBR

This function controls the timing parameter at run time. If the argument is non-NIL, then subsequent top level evaluations will be followed by a message giving the elapsed time of their execution in microseconds.

*TIME: number

If the argument is NIL, the timing messages will be suppressed. The value of timing is NIL.

backtrace[]

pseudofunction; SUBR

This function returns a list of the current indicators on the system stack. See Chapter 8 for a description of these items.

gcmess[s]

pseudofunction; SUBR

This function simulates the G control card parameter at run-time. If the argument is non-NIL, the garbage collector message will be enabled. If it is NIL, the message will be disabled. The value of gcmess is NIL.

listing[s]

pseudofunction; SUBR

This function simulates the L, P, and N parameters of the LISP control card. If the argument is the atom P, then subsequent top level expressions or doublets will be listed with parenthesis counting. If the argument is the atom L, then subsequent top level expressions or doublets will be listed without a parenthesis count. If the argument is the atom N, then only user-originated output and error messages will appear on SYSOUT. If the argument is anything else, then subsequent top level expressions and doublets will be printed by LISP but not in source image form. The value of listing is always NIL. This function has no effect if SYSIN and SYSOUT are the same actual file.

4.13 ARRAYS

Although the primary data structure of LISP is the linked list, an array structure is useful in many problems. UT LISP provides a primitive array capability. Array storage is allocated in binary program space (see 3.) with two array elements per word. Each array element can be a pointer to an arbitrary S-expression and is preserved during garbage collection. Memory allocated to an array cannot be reused by LISP for other purposes.

`mkarray[litatom;(integer1,integer2,...integern)]`

pseudofunction; SUBR

mkarray defines and allocates an n-dimensional array whose "name" is litatom. Each integer_i is a dimension of the array such that elements of the array are indexed in the i-th dimension by values of 0 to (integer_i - 1). A pointer to the array space is placed on the property list of litatom with the indicator ARRAY. Each array element is initialized to NIL. The value of mkarray is NIL. The total number of elements in arrays is $\prod_{i=1}^n \text{integer}_i$.

`setel[litatom;(integer1,integer2...integern);s]`

pseudofunction; SUBR

If litatom has an ARRAY property, setel stores the S-expression S into the element indexed by the list of subscripts in the second argument. Each subscript may range from 0 to d_i - 1 where d_i is the value of the i-th dimension used in the call to mkarray which allocated the array. If fewer subscripts are used than the dimensionality of the array calls for, the effect is as though they were omitted from the right and values of 0 are used in their place. e.g., setel[a;(1);NIL] is equivalent to setel[A;(1 0);NIL] if A is a two-dimensional array. The value of setel is S, the third argument. In the interest of speed of access no validity checking is performed on the subscript list.

getel[litatom; (integer₁, integer₂...integer_n)]

pseudofunction; SUBR

If litatom has an ARRAY property, getel returns the value stored in the element indexed by the list of subscripts in the second argument.

All conventions regarding treatment of subscripts are the same as defined for setel above.

CHAPTER 5

INPUT/OUTPUT

LISP provides an extensive set of functions which give the user facilities for input and output of LISP S-expressions and general data on any disk file within the operating system. With these functions the user can perform rather complicated file manipulation operations. Functions described in this chapter will be described using the same notations as in Chapter 4.

5.1 FILES

All input/output in the operating system is done via files of information. LISP regards a file as a contiguous string of lines with an end-of-file mark and a position pointer. Although the operating system can structure files into logical records LISP completely ignores this structure and end-of-record marks are invisible to the LISP input/output system. Files are named and their names are represented within LISP by literal atoms whose print representations are character strings of 7 characters or less. Within LISP three file names are used for special purposes and cannot be used for disk file names. They are:

<u>File name</u>	<u>Usage</u>
TTY	Communication with the teletype.
CEE	Used by a controller program to communicate with its controllee.
CER	Used by a controllee program to communicate with its controller.

All data read or written by LISP input/output functions is in the form of packed ASCII line images obeying the usual system conventions. The input routines assume line images of 72 columns or less. If a line being read is longer than 72 columns all information beyond column 72 will be ignored by the LISP system. Under program control the line length allowed for output lines is changeable.

LISP keeps a record of the last type of operation performed on a file and will always perform a rewind operation any time a read is requested when

the last operation was a write. Thus the user can write information on a file and immediately read that file from the beginning.

If LISP attempts to write on a disk file that does not exist, it will create a type DATA file of that name before writing. The file will be created with the length specified by the SIZE parameter of an open call (see sec. 5.2) or with a length of 157700₈ if no other size has been specified.

If LISP attempts to read a disk file that does not exist, it will type out an informative message and pause. If the user can make the file available, he should do so and then type "GO" to allow LISP to continue. Typing anything else will cause LISP to terminate.

If LISP attempts to write on a read-only file it will ask the user to give it permission to change the file type to read/write. The user should type "RW" to grant permission. Typing anything else will terminate the run.

5.1.1 STANDARD SYSTEM INPUT/OUTPUT FILES

At any time several files may be known and usable by LISP. Two of these files are designated as the standard system input/output files. The standard input file is the one from which the LISP main interpreter loop reads the program expressions or doublets. The standard output file is the one on which the LISP main interpreter loop writes the results of expression or doublet evaluations and also all error messages.

These files are known by the pseudonyms SYSIN and SYSOUT, respectively. SYSIN and SYSOUT are each equivalenced to some other file which is the file actually read or written. This equivalencing is under program control. No matter what files are being used for the standard system input/output, they may be referenced by the names SYSIN and SYSOUT.

If not otherwise specified, SYSIN is initially equivalenced to INPUT and SYSOUT is initially equivalenced to OUTPUT. The input file control and output file control parameters on the LISP control card can be used to initialize SYSIN and SYSOUT to other file names. The conversational mode control parameter equivalences both SYSIN and SYSOUT to file TTY.

SYSIN and SYSOUT can be changed by means of the function below:

`sysin[filename; litatom]`

pseudofunction; SUBR

sysin makes filename the new standard system input file. The next expression or doublet will be read from filename. sysin makes a stack of filenames given in successive calls to sysin so that when the end of the new SYSIN is read the stack is popped and SYSIN reverts to the last previous value. Thus sysin may initiate the input of a whole chain of files, with SYSIN eventually returning to its first definition. The function sysin returns as value the name of the old SYSIN-equivalenced file. The second argument of sysin provides listing control for the new SYSIN file. If the litatom is L, P, or N, the new SYSIN file will be listed or not as though the L, P, or N parameters appeared on the LISP control card (see section 2.1). If the litatom is NIL, the listing controls in effect at the time sysin is called will be in effect for the new SYSIN file. If the litatom is anything else, all listing control flags will be cleared.

`sysout[filename]`

pseudofunction; SUBR

sysout makes filename the new standard system output file. It takes effect immediately, and the value of sysout will be written on filename. The function sysout returns as value the name of the old SYSOUT-equivalenced file.

5.1.2 SELECTED READ AND WRITE FILES

At all times one file is designated the selected read file and one the selected write file. LISP functions which read information generally read from the current selected read file and those functions which write information generally write on the selected write file.

Initially, the selected read and write files are the same as SYSIN and SYSOUT, respectively. Execution of functions sysin and/or sysout has no effect on the selected read and write files, however. These files may be changed by functions rds and wrs as described below.

`rds[filename]`

pseudofunction; SUBR

rds makes filename the new selected read file. It returns the previously-selected read file name as its value.

`wrs[filename]`

pseudofunction; SUBR

wrs makes filename the new selected write file. It returns the previously-selected write file name as its value.

The user may at any time determine which files are currently selected by executing the function sfiles.

sfiles[]

pseudofunction; SUBR

sfiles returns a list of the form:

```
((SYSIN . filename1)(SYSOUT . filename2)(RDS . filename3)
(WRS . filename4))
```

where the filenames are the names of the present standard system input/output and selected read and write files.

5.2 FILE AND BUFFER ASSOCIATIONS

With each file used in LISP there must be a buffer area associated with that file so long as it is in use. The standard LISP system includes six file buffer areas which may accommodate up to six active files at one time. It is not possible for the user to access more than six files at any given time. Under program control, however, the file buffers may be detached from files and attached to other files so that a total of more than six files may in fact be used during the course of a run.

The act of associating a buffer with a file and defining certain characteristics of the file is called "opening" the file. Unless stated otherwise, all functions described in other sections of this chapter "open" the files they manipulate using default values for the file characteristics. If the user wishes to explicitly "open" a file or define its characteristics, the function open is used.

open[filename; ((c₁ . v₁)(c₂ . v₂)... (c_n . v_n))]

pseudofunction; SUBR

open finds an available buffer and associates it with filename. open returns filename as its value. The second argument of open is a list of file characteristics to be set. They are chosen from among the following:

<u>Characteristic Name</u>	<u>Value</u>	<u>Meaning</u>	<u>Default Value</u>
ECHO	NIL or non-NIL	If value is non-NIL, then each line of the file is echoed on SYSOUT as the file is read. NIL turns this off.	NIL
ECHOP	NIL or non-NIL	If value is non-NIL, each line of the file will be echoed on SYSOUT as the file is read, and a parenthesis count will also be printed for each line. NIL turns this option off.	NIL
LENGTH	Number	The maximum output line length for this file is <u>number</u> characters.	70
MARGIN	Number	Each line output to this file will have <u>number</u> blank characters appended to the left. MARGIN plus LENGTH should not exceed 136.	0
SCR	NIL or non-NIL	Defines file as scratch mode if non-NIL. Scratch mode files are never written to disk so long as the information written on them does not exceed the capacity of the buffer. Scratch mode is cancelled if the buffer overflows or if the function endfile is executed. For short files which do not need to be preserved at the end of a run, scratch mode significantly speeds access.	NIL
SIZE	Number	Defines the size at which the file is to be created when the first write is attempted. This has effect only if the file does not yet exist at the time of the call on <u>open</u> .	157700 ₈

Any characteristics not mentioned in the second argument to open will be defined with the default value. open may be executed any number of times on any file to dynamically change the file characteristics.

The act of disassociating a file from a buffer is called "closing" the file. The function close can be used to disassociate a file and its buffer.

close[filename]

pseudofunction; SUBR

close disassociates the file named from its buffer. The filename is returned by close as its value. After the buffer has been disassociated from its file it is available for re-use for some other file. If the file being "closed" was last used for output purposes, an end of file mark is written on the file. Any "closed" file still exists on the disk and may be "opened" again later.

The user may determine what files are "open" at any time by executing function openfiles.

openfiles[]

pseudofunction; SUBR

openfiles returns a list with one element for each file presently "open" to LISP. Each element is a list of the form:

$$(\text{filename}((c_1 . v_1)(c_2 . v_2) \dots (c_n . v_n)))$$

where each of the $(c_i . v_i)$ pairs indicates the current value of one of the characteristics of the files.

5.3 OUTPUT OF S-EXPRESSIONS

The functions print, write, prinl, and terpri are used to output LISP S-expressions on the current selected write file. Function output can be used to output an S-expression to an arbitrary file. The S-expressions will be written on as many lines as are required, governed by the margin and line length of the file being written. Formatting of numeric atoms is completely automatic and may be controlled to some extent by use of the function nformat (see 5.8). If the output functions should happen to be given something to write which is not an S-expression, they will write in its place

↪nnnnnn

where nnnnn is the octal value of the actual pointer received by the output function.

```
print[s1;s2]  
write[s1;s2]
```

pseudofunction; SUBR

The value of print is its first argument. print writes the S-expression s₁ on the current selected write file. Writing commences on the current line of that file and extends over as many lines as necessary. print always terminates the line so that the next S-expression written on the file will begin on a new line. If the second argument of print is NIL, the S-expression will be printed using the print names of its component atoms. If the second argument is non-NIL, any atoms with nonstandard print names will have the appropriate delimiters inserted so that they may subsequently be read by the LISP input functions. write is a synonym for print.

`prinl[s1;s2]`

pseudofunction; SUBR

prinl behaves identically to print except that it does not terminate the line after writing the S-expression. Successive calls on prinl will cause several S-expressions to (possibly) be printed on one line, side by side.

`terpri[]`

pseudofunction; SUBR

The value of terpri is NIL. terpri terminates the current line on the current selected write file so that the next S-expression written on the file will begin a new line. A call on prinl followed by a call on terpri is equivalent to a call on print. If nothing has been written on the current line, the effect of terpri is to skip a blank line.

`output[filename; s1;s2]`

pseudofunction; SUBR

The value of output is its second argument. output writes its second argument on the file given as its first argument. With respect to that file, the action of output is the same as print. output does not change the selected write file. The definition of output is equivalent to:

```
output[filename; s1; s2] =  
  prog[(A); setq[A; wrs[filename]];  
    print[s1; s2]; wrs[A]; return[s1]]
```

5.4 INPUT OF S-EXPRESSIONS

The functions read and input are used to read LISP S-expressions from files. Each call of one of these functions reads the next available complete S-expression on that file, consuming as many lines of the file as necessary. When the complete S-expression has been read, the file is left positioned so that the next read request will continue scanning the line on which the previous S-expression ended. Thus there may be more than one S-expression on a line. No more than 72 columns of each line are scanned.

read[]

pseudofunction; SUBR

read reads the next S-expression from the current selected read file and returns that S-expression as its value. If there is no next S-expression on the file the atom \$EOF\$ is returned as the value of read. If the end-of-file is detected while reading a list, an "unmatched left parenthesis" error will occur.

input[filename]

pseudofunction; SUBR

input operates in the same manner as read except it reads directly from the file, filename. input does not change the selected read file. The definition of input is equivalent to:

```
input[filename] = prog[(A X); setq[A; rds[filename]];  
  setq[X; read[]]; rds[A]; return[X]]
```

5.5 INPUT OF NON-S-EXPRESSIONS

It is often necessary to read information which is not in the form of S-expressions. LISP provides very primitive facilities for reading such data one character at a time. It is up to the user to employ the various character-manipulating functions described in Section 4.10 to reform the data into meaningful objects. Characters read are returned by these functions as the literal atom which has that single character as its print image. Users should carefully note that digits (0,1,2,...,9) read as characters are returned as a literal atom equivalent to (\$\$0\$, \$\$1\$, ..., \$\$9\$) and are consequently not equivalent to the single-digit numbers. The function numob must be used if it is desired to read numbers this way.

Users should also note that line images as stored on files by the operating system are variable in length. The operating system truncates all lines by removing trailing blanks, leaving an even number of characters followed by an end-of-line marker. LISP considers the end-of-line to occur at column 73 or when the end-of-line marker is encountered, whichever occurs first. Therefore users should not use column counts to determine when lines are terminated.

readch[s]

advance[s]

pseudofunction; SUBR

readch (advance is a synonym) returns the next character from the current selected read file if its argument is NIL, with each such call the input position is advanced one column. If the argument of readch is non-NIL, the input position will be backed up two columns before reading to allow the previous character to be read again. This backing-up function of readch cannot go beyond the beginning of the current line and any attempt to do so will cause the atom \$EOR\$ to be returned as the value of readch. \$EOR\$ is also returned by readch when it detects the end-of-line. If the end-of-line is reached, the next call on readch will automatically sequence the input position to the beginning of the next line and the first character on that line will be returned by readch. The atom \$EOR\$ is returned by readch if it advances the input position beyond the last line of the file.

startread[]

pseudofunction; SUBR

startread moves the input position to the beginning of the next line of the current selected read file whether or not the current line has been entirely scanned. The value of startread is the first character on the new line. Subsequent calls on readch will continue scanning the new line. Contrary to some LISP implementations, UT LISP does not require startread to be called before the first call on readch. startread returns the atom \$EOF\$ if there is no next line on the file.

teread[]

endread[]

pseudofunction; SUBR

teread (endread is a synonym) immediately advances the input position to the end-of-line of the current line of the current selected read file. The value of teread is always the atom \$EOR\$. After a call on teread, the next call on readch will read the first character on the next line.

5.7 INPUT CONTROL FUNCTIONS

UT LISP provides a limited amount of control over the input process. Functions are provided for skipping input columns or for tabbing to a particular column.

`ispace[number]`

pseudofunction; SUBR

ispace positions the input pointer of the current selected read file forward by number columns. ispace will not skip beyond the actual end of the current line. The value of ispace is the column number of the next column to be read after the skipping has been done. If the argument to ispace is ≤ 0 no skipping is done. Thus ispace [0] will give the present column position on the input line.

`itab[number]`

pseudofunction; SUBR

itab positions the input pointer of the current selected read file forward to the column specified by its argument. If the present column number is greater than or equal to the argument of itab, no positioning is done. The value of itab is the number of the next column to be read after the positioning has been done. itab will not move the input pointer beyond the actual end of the present line.

5.8 OUTPUT CONTROL FUNCTIONS

UT LISP provides a limited amount of control over the output process in addition to that previously described. Functions are provided for intraline spacing and for formatting floating point numbers.

`ospace[number]`

pseudofunction; SUBR

ospace issues number blanks to the current line of the current selected write file starting at the current column position. ospace will not space beyond the defined line length of the file. The value of ospace is the number of the output column in which the next output item will begin. Arguments of ospace which are ≤ 0 will not cause spacing to be performed. Thus ospace[0] will give the current column number as its value.

`otab[number]`

pseudofunction; SUBR

otab positions the current selected write file so that the next item output will begin in the column number specified by the argument to otab. otab will not space beyond the defined line length of the file. Arguments of otab which are less than the current column position will not cause spacing to occur. The value of otab is number of the output column in which the next item output will begin.

`nformat[number1; number2]`

pseudofunction; SUBR

The first argument to nformat is made the number of digits preceding the decimal point when floating point numbers are subsequently output; the second argument specifies the number of digits to follow the decimal point. Both arguments must be integral and their sum cannot exceed 16. If their sum does exceed 16 the second argument will be reduced if possible. The details of how number-formatting is determined are given in sec. 3. -. The standard values of the two numbers are 1 and 10 respectively. The value of `nformat` is NIL.

Example:

The number 23.123 is printed as

2.3123000000E+1

under the standard format. After executing `nformat[2;2]` it will print as

23.12

5.9 FILE MANIPULATION

The remaining input/output functions of UT LISP are those for file manipulation. These provide for the usual cases.

`abolish[filename]`

pseudofunction; SUBR

abolish closes the file given as its argument and then returns the file to the operating system, releasing the disk space or tape unit assigned to the file. That file can no longer be referenced. The value of abolish is its argument.

`endfile[filename]`

pseudofunction; SUBR

endfile writes an end-of-file mark on the named file. If the file was opened as a scratch-mode file, endfile cancels the scratch-mode and forces the file to be written on disk. If subsequent writes are made on a disk file the end-of-file mark is changed to a system end-of-record mark. The value of endfile is its argument.

`rewind[filename]`

pseudofunction; SUBR

rewind causes the named file to be repositioned at its beginning. If the file is already rewound, it is not disturbed. The value of rewind is its argument.

CHAPTER 8

DEBUGGING THE LISP PROGRAM

This chapter explains the causes and cures of the errors detected by UT LISP and offers genral advice for debugging LISP programs. In the interpretive mode, LISP has very powerful error-checking capabilities, and the associated messages can provide the user with considerable information to aid in debugging his program. Extensive error checking is not provided in compiled LISP code in order that it may run faster; it is assumed that compiled functions have been thoroughly debugged before compilation.

8.1 ERROR MESSAGES

FIELD TOO SHORT FOR LISP

is a message given when the LISP system finds that the field length it was loaded into provides an impractically small amount of free and full-word storage. A field length of 50000_g is recommended as the minimum practical size.

8.2 UT LISP ERROR MESSAGES AND THEIR INTERPRETATION

Three types of error messages may appear in the course of a LISP run:

- (a) Informative messages. These messages report conditions which are not fatal to execution but should demand user attention. After printing the message, the LISP system continues with the program as best it can. These messages are preceded by `/////`.
- (b) Recoverable errors. Errors represented by these messages cause execution of the current expression or doublet to be terminated. The LISP system is able to recover, however, and continue with the next expression or doublet. These messages are preceded by `*****`.

All error messages are set off from the rest of the output by blank lines before and after, and therefore should be easily spotted by the user.

Most recoverable error messages are accompanied by a backtrace. Informative messages are not so accompanied. The backtrace is a reduced listing of the contents of the LISP stack at the time of the error, and the primary information it conveys to the user is the sequence of functions which have been entered, but not completed. In form the backtrace is:

```
>>>>>STACK: (name name name ... name)
```

for as many lines as are necessary. Occasionally, information other than functions names, such as lists and numbers may appear, but the chief fact of interest to the user is that every function entered, both user-defined and system-defined, will have its name on the stack. The names are printed in forward order, with the most-recently-entered function appearing last. (See Section 8.4.)

8.2.1 ERRORS DETECTED DURING INPUT

These errors imply that the user's data is incorrect. They can be eliminated only by revising the data.

```
///// ILLEGAL DOTTED PAIR SYNTAX
```

While reading an S-expression, LISP detected one of the forms

```
--- S-exp1 . S-exp2 . S-exp3)
```

```
or --- S-exp1 . S-exp2 S-exp3)
```

Only --- S-exp₁ . S-exp₂) is a legal dotted pair. S-exp₃ is completely ignored, and LISP continues. If the file is being listed as it is read, this message will appear directly beneath the line containing the error.

```
///// ILLEGAL SEQUENCE .)
```

```
///// ILLEGAL SEQUENCE ( .
```

These two character sequences are illegal in an S-expression. The period is ignored in both cases. If the file is being listed as it is read, these messages will appear beneath the offending line(s).

///// CONSECUTIVE DOTS

Two or more consecutive periods were found in a dotted pair. All but the first period will be ignored. If the file is being listed as it is read, this message will appear beneath the line containing the error.

///// UNMATCHED RIGHT PARENTHESES

This message occurs if the first non-blank character encountered when a new S-expression is read is a right parenthesis. The implication is that there is an excess of right parentheses over left parentheses. The extra parentheses are ignored. If the file is being listed as it is read, this message will appear directly beneath the offending line.

///// EXTRANEIOUS DOTS

This message occurs if the first character read for an S-expression is a period. That is, one or more periods appear between S-expressions on the input file. The periods are ignored. If the file is being listed as it is read, this message will appear directly beneath the offending line.

***** UNMATCHED LEFT PARENTHESES

This message occurs if an end-of-file is read before sufficient right parentheses have been read to match all left parentheses. It is symptomatic of missing data cards.

8.2.2 ERRORS DETECTED DURING OUTPUT

See Section 5.5.

8.2.3 ERRORS DETECTED BY FILE MANIPULATION FUNCTIONS

***** ATTEMPTED TO OPEN MORE THAN SIX FILES

This error occurs for calls to the rds, wrs, rewind, endfile, or open functions which attempt to open a seventh file. LISP can only accomodate

six files at one time. If more than six files are needed the advised approach is to close a file and open it again later if necessary.

***** ATTEMPTED TO CLOSE SYSIN OR SYSOUT

This error is detected by the close function. It is never permissible to close the file which is equated to SYSIN or the one equated to SYSOUT. SYSIN and SYSOUT must always exist. They may be changed, however. (See Chapter 5.)

8.2.4 ERRORS DETECTED BY THE GARBAGE COLLECTOR

***** INSUFFICIENT FREE SPACE

***** INSUFFICIENT FULL SPACE

These errors occur when the system finds that the amount of space recovered in either of the two areas is less than 1/64th of the space initially allocated to the area. The implication is that it would be futile to continue with execution as garbage collection would occur more and more frequently with less and less gain. When either of these two messages occur, the system will return control to the last occurrence of errorset or to mainloop, whichever occurs first.

The problem

can be cured most easily by increasing the memory size used in the next run. Alternatively the A parameter on the LISP control card may be adjusted (see Chapter 2). Decreasing the A parameter effectively allocates more of the given space to the full-word space region, whereas increasing the A parameter allocates more of the given space to the free space.

///// FREE SPACE IS CROWDED

///// FULL SPACE IS CROWDED

These two informative messages occur if the garbage collector finds that less than 1/16th of the allocated space in the respective region is available. They are an indication that the program may run more efficiently in either a larger memory space or else with adjustment in the A parameter on the LISP control card (see Chapter 2).

///// GARBAGE COLLECTION:

This informative message appears every time a garbage collection occurs if the G parameter was present on the LISP control card, or if it is turned on by the gcmess function (see Section 4.11). Immediately following the message will appear two numbers, the first of which is the number of full-space words which were recovered and the second is the number of free-space words which were recovered.

8.2.5 ERRORS DETECTED BY THE INTERPRETER

***** UNDEFINED FUNCTION:

The interpreter has encountered an atom which is syntactically in the position of a function name for which it was unable to find a functional definition. Immediately following the message will be the name whose

definition was not found. Most commonly this error is caused by an extra set of parentheses around a form within a LISP expression. Such a pair of parentheses causes the LISP interpreter to treat the results of the form inside as another call on a function. Typically, the result will be some data list whose first element is not a function name. Another cause is the omission of a necessary quote in front of a data list which will cause the data list to be interpreted as a form. Finally, this error may arise if there was a parenthesis error within the expression or doublet which (was supposed to have) defined the function. Such an error may cause the function not to be recognized by define or deflist. The backtrace will indicate the function within which the erroneous function call occurred.

***** NUMBER TREATED AS FUNCTION:

This error occurs when the interpreter finds a number in a position where it expects a function name. The causes of the error are the same as for the "undefined function" error. That is an extra set of parentheses around a form or else a missing quote. Again, the backtrace would indicate which function contained the erroneous information. The number itself will be printed after the message.

***** RECURSION LIMIT EXCEEDED:

Every time a function is entered certain information is placed on the stack and is removed from the stack when the function is exited. Typically the amount of information placed on the stack is from two to six words. This error occurs when there is no more room on the stack. It is impossible to specify how many levels of recursion are allowed by the system as it depends on the particular sequence of functions which are entered. In general, though, it would appear that a stack size of one thousand words would allow two hundred to three hundred levels of recursion. Technical reasons for this error to occur are the failure of the programmer to include terminating conditions in a recursive function or the error may occur in certain system functions such as print and append if they are given a circular list to process. Finally, of course it is possible that a recursive function is operating on data which causes it to recur too many times. If the latter is the case, then it is advised that that function be rewritten

in an iterative manner using the PROG feature. This problem can also be eliminated by increasing the stack size via the B parameter on the LISP control card. No backtrace is given with this error. The name of the last entry on the stack is, however, printed following the error message.

***** UNBOUND VARIABLE:

A variable which has not been assigned a value can be detected by the LISP system. This message will occur when the value of such a variable is required in a computation. Immediately following the message will be the name of the supposed variable and in the backtrace will be information about which function it was encountered in. Typical causes for this error are keypunch error or a missing quote around a data atom in which case the data atom would be evaluated as a variable. This error is not detected within compiled functions.

***** TOO MANY ARGUMENTS:

***** TOO FEW ARGUMENTS:

These two messages occur when a user defined EXPR function is called and given either too many or too few arguments. These conditions are not detected for machine-coded SUBRs or compiled functions. The name of the called function appears immediately after the error message and the name of the calling function will appear in the backtrace information. The user must re-define his functions to correct this error.

8.2.6 ERRORS DETECTED WITHIN PARTICULAR LISP FUNCTIONS

***** ILLEGAL ARGUMENT:

This error is detected by a number of LISP functions. Immediately following this message the system will print cons [fn;arg] where fn is the name of the function and arg is the offending argument. The backtrace will give the name of the function within which the call was made. The user should consult the function descriptions (Chapters 4 and 5) to determine the reasons for the error. The error typically occurs when a function is given an atom when a list is expected or vice-versa, or else when a numeric function is given a non-numeric argument. At times the function name given may not seem to be related to the function called by the user. This will happen when certain of the LISP functions defined in the system call on other functions internal to their operation.

***** ILL-FORMED ARGUMENT:

This message is similar to the illegal argument error. It occurs when a non-list dotted pair is given to a function which expects a list and looks for the nil terminator of that list. The name of the function receiving the ill-formed argument appears immediately following the error message. The user must re-write his call on the function to correct the error.

***** ERROR:

This message occurs for a user-detected error when the user calls the system error function. Immediately following the message will be an S-expression. This S-expression is the argument to the error function and can be any message that the user desires.

***** RETURN OR GO OUTSIDE A PROG

This error occurs if a return or go function was called within another function which was not executing within the overall control of a prog expression. This situation is simply not allowed and the user must re-define the function to correct the error.

***** GO TO NON-EXISTENT LABEL:

Within a prog expression the function go was called with an argument which was a non-existent label. Immediately following the error message will be printed the name of the label in question. The most common cause of this error is keypunch error.

***** CHARACTER BUFFER EXCEEDED

Detected by the pack function, this message implies that too many characters were attempted to be packed into the character buffer. The character buffer has a maximum limit of 120 characters.

8.3 WHAT TO DO IF THE ANSWER IS WRONG

It is, of course, often the case that a program will run and none of the errors previously described will happen, but yet the results provided by the program will be incorrect. In such a case it will be necessary for the user to apply a debugging procedure to determine the errors in his own functions. We shall try here to develop a reasonable procedure to follow for debugging running LISP programs.

It is assumed that the user is familiar with the purposes of each of the functions that he is attempting to use. If this is true, then it is often possible by studying the form of the incorrect result to determine approximately within the set of functions where the error is occurring. This will greatly reduce the amount of effort required in the debugging process as the user can focus his attention in a specific area. In any event the first step is to carefully review the form of the functions as they were input to the LISP processor. The easiest way to do this is to obtain a listing of the functions showing the parenthesis level count. This can be obtained using the P parameter on the LISP control card (see Chapter 2). With such a listing, the user should carefully scrutinize his function definitions and be certain that they are syntactically correct. It is possible for simple parenthesis errors to be made which will cause the functions to have an incorrect syntactical form so that they will be correct overall but yet not serve the intended purpose and not cause a LISP system error. Any errors in syntactic form discovered in this way should be corrected, and the job rerun to determine if the erroneous answer still occurs.

Once the syntactic form of the user's functions is correct and an erroneous result still occurs, then it is necessary to use the various tracing facilities of the LISP system to gain more information about the behavior of the functions. The trace function is described in Chapter 4 and can be used to turn on tracing for any selected set of functions, both user functions and system-defined functions. Its converse, the untrace function, can be used to selectively turn off such tracing. Once tracing of a function is turned on then every time that function is entered the arguments which were actually received by it will be printed, and the value it generates will also be printed when it is exited. The arguments and value of a function are printed every time it is entered and left even if it is entered via recursion from within itself. Thus, tracing a recursive function will yield complete information about the way it operates on a given set of data.

Tracing of functions should be used very carefully. The extra printing involved in the tracing information costs the user both pages of output and extra time. If a great many functions are traced at one time, these extra costs can add considerably to the cost of a job. It is suggested, therefore, that the user trace only those functions of which he suspects erroneous operation. Also, he should trace those functions while they are operating on a minimal-sized data set which will reproduce the error. A very useful procedure to apply is to make one run tracing all of the functions in which a suspected error is occurring; then, by reviewing the output, the user can determine which of the traced functions are operating correctly, and perhaps he can determine which of the functions is operating incorrectly. If the erroneous function(s) cannot be determined for one reason or another, the tracing output will usually point to some other functions which may not have been traced that are better candidates to be traced the next time. Then the next test run should be made without tracing those functions which are known to be operating correctly. Thus, through a progressive series of runs the erroneous function can be isolated.

Tracing, as has just been described, is very useful for debugging recursive functions. It does not, however, always yield complete information about iterative functions which have been written using the prog feature. In this case, tracing shows the arguments which were received by the function; and shows the value which was returned by the function; but shows nothing of what happened in between, unless the iterative function is also recursive. For

such functions another tracing procedure is available, namely the traceset function. This function and its converse, the untraceset function, have both been described in Chapter 4. They allow one to selectively turn on and turn off the tracing of all set and setq value assignment statements within a prog expression. When traceset has been applied to a given function then every time a variable is assigned a value via the set, or setq, functions within the prog expression (or in any recursive function which is executed under the control of the prog expression) a line will be printed showing the variable and the value which was assigned to the variable at that time. Since iterative functions use variables to hold temporary results and as counters, etc., the traceset option allows one to obtain almost complete information about an iterative function. The same warning offered with respect to the trace function can also be given in respect to the traceset function.

By judicious usage of the trace and traceset functions the user can normally determine very closely the point at which an erroneous answer is being generated. In the unlikely event that the error appears to be occurring within one of the system-defined functions, the user should report such a finding to one of the Computation Center's consultants showing him the output necessary to support the conclusion.

Tracing is not always as flexible as might be desired. For instance, it might be the case that it is desired to trace a certain function only when it is called by another function. As now implemented the trace will cause the arguments and value to be printed every time the function is called, and not just when it is called by the desired function. In such a case the user is recommended to place the call of the trace function within the function he wishes to trace from, and to subsequently follow that with an untrace call to turn off the tracing for calls within other functions. Or the user might wish to actually program the printing of intermediate results. This is quite easily done in LISP since the print function is an identity function. That is, it returns its argument unchanged as its value. Therefore, any form which appears in the LISP program may be made the argument to print, and the result of the program will be unchanged.

To reiterate we shall state the debugging procedure as a step-wise process.

- (1) From a listing with parenthesis counts check for and correct syntactic errors.
- (2) Use trace and/or traceset to selectively trace the functions which are believed to be in error, and progressively isolate the functions which actually are in error.
- (3) Correct the erroneous function(s).

8.4 INTERPRETATION OF THE UT LISP BACKTRACE

The backtrace is a condensed listing of the content of the system's pushdown stack. It contains a history of the activity of the system up to the point in time at which the backtrace is taken.

Of most interest is the fact that the names of all functions which were entered but not yet completed will appear. Often these names will appear twice side-by-side because either eval or apply put the name on and the function itself put its own name on. The user should not be alarmed by this.

The special name VMARK appears frequently, although it is not a function name. This symbol is placed on the stack each time a group of variables is bound. Such binding occurs on entry to any lambda expression and when binding prog variables. The sequence PROG VMARK PROG appears when a prog expression is entered.

Other symbols, particularly APPLY, EVAL, and EVLIS appear even though the user did not call these functions since they are used internally by the LISP interpreter. If the backtrace is taken while a series of function arguments is being evaluated, then the already-evaluated arguments will appear on the stack.

8.5 PROGRAM-DETERMINATION OF ERROR TYPE

APPENDIX A

CHARACTER TRANSLATION TABLE FOR UT LISP

<u>Character</u>	<u>ASCII</u>	<u>DPC</u>	<u>Character</u>	<u>ASCII</u>	<u>DPC</u>
blank	00	55	@	40	74
!	01	00	A	41	01
"	02	60	B	42	02
#	03	64	C	43	03
\$	04	53	D	44	04
%	05	66	E	45	05
&	06	67	F	46	06
'	07	76	G	47	07
(	10	51	H	50	10
)	11	52	I	51	11
*	12	47	J	52	12
+	13	45	K	53	13
,	14	56	L	54	14
-	15	46	M	55	15
.	16	57	N	56	16
/	17	50	O	57	17
0	20	33	P	60	20
1	21	34	Q	61	21
2	22	35	R	62	22
3	23	36	S	63	23
4	24	37	T	64	24
5	25	40	U	65	25
6	26	41	V	66	26
7	27	42	W	67	27
8	30	43	X	70	30
9	31	44	Y	71	31
:	32	63	Z	72	32
;	33	77	[	73	61
<	34	72	\	74	71
=	35	61	]	75	62
>	36	73	↑	76	70
?	37	75	←	77	65

APPENDIX B

MISCELLANEOUS ERROR MESSAGES AT LLL

SYSTEM CALL ERROR: ADDRESS

AN UNRECOGNIZABLE REQUEST WAS MADE TO THE OPERATING SYSTEM AT ADDRESS.

FAILURE IN DROP FILE CREATION

SELF-EXPLANATORY.

RFL FAILURE

FLOE FAILED TO ADJUST FIELD LENGTH.

RFL REQUEST TOO LARGE

LISP REQUESTED TOO MUCH SCM FROM FLOE.

CIO ERROR N

AN ILLEGAL I/O OPERATION WAS REQUESTED. VALUES OF N AND THEIR MEANINGS ARE:

- 1 - UNRECOGNIZED OPERATION
- 2 - FLOE FAILED IN TRYING TO CLOSE A FILE
- 3 - FILE EXISTS, BUT COULD NOT BE OPENED.
- 4 - REWIND REQUEST ON NONEXISTENT FILE
- 5 - REQUEST MADE FOR UNIMPLEMENTED RANDOM OPERATION
- 6 - ATTEMPTED READ AFTER WRITE

*****CATASTROPHE*****

CONTROLLER - CONTROLLEE COMMUNICATIONS CONFUSION.

DISTRIBUTION

UCID-16650
Page 85

LLL Internal Distribution

TID File

30

External Distribution

Mark Blair

UC - Berkeley, CA.

USAEC, Technical Information Center,
Oak Ridge, Tennessee

2

NOTICE

"This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately-owned rights."