LISP/66 Users Manual
Version 2.3

David G. Conroy

Department of Computer Science
University of Waterloo

Report CS-76-12

March 1976

Chapter 1
Introduction

LISP/66 is an interactive LISP interpreter designed to run under TSS/GCOS on Honeywell/66 series computers (LISP/66 requires a release E or later GCOS system. It will run under older releases but does all its disc file input/output in media 6 ASCII, which the old ASCASC sybsystem does not understand). In addition to providing most of the standard functions of other LISP systems, LISP/66 offers improved character handling facilities and a file oriented input/output system.

This manual is intended to outline the behaviour of the LISP/66 system to programmers already familiar with the LISP language. Readers wishing to learn LISP should first read one of the introductory texts.

1.1 Getting Online

LISP/66 is invoked by typing the LISP command at system level. LISP will respond by prompting for user input with a question mark (in the following and all other examples, system output is capitalized)

```
SYSTEM ? lisp
?
```

LISP/66 is now in a listen loop - reading two s-expressions, passing them to EVALQUOTE, and printing the returned value.

```
?car((a b c))
A
?cdr((a,b,c))
(B C)
?cons
? (a
? b
? )
(A . B)
```

Note that all user input is mapped into upper case, and that tabs and carriage returns may be used to delimit atomic symbols (as well as the usual blanks and commas).

If LISP/66 gets hung up (due to the loop in a function, for example) the break key may be used to return control to the EVALQUOTE listen loop.

Typing *done* to the listen loop will terminate LISP and return the user to system level. A function called *done* will be impossible to call from top level, and removing the atom *done* from the object list will make it impossible to exit in a normal manner.

1.2 Command Line Options

The preceeding TSS dialog causes LISP/66 to allocate default quantities of storage. Currently these are set to 4K words free space (which can dynamically grow) and no binary program space (which cannot grow). In order to specify initial storage allocations command line options must be specified.

```
SYSTEM? lisp [-bv] [c=n] [b=n] [l=n]
```

-b change the default prompt (question mark) to an ASCII Escape. This option is useful to supress the printing of the control Q on LSI ADM series terminals.

-v display the system version number on entry.

c=n specifies the initial size of free storage in units of 1024 words. "n" is a decimal integer.

b=n specifies the initial size of the binary program space in units of 1024 words. "n" is a decimal integer.

l=n specifies the initial output line length. "n" is a decimal integer. The default line length is 80 characters.

1.3 System Messages

PDS OVERFLOW - The internal stack has overflown. Functions recursing without end or deeper than about 2000 levels will cause this message.

COLLECTING - This message is generated whenever the garbage collector is invoked to create a new free storage list.

9+nnK CORE - This message is generated when LISP/66 expands its free storage region; nn is the new size in K words. Both this and the above message are controlled by the *verbose* toggle function (see Section 2.10)

SYNTAX ERROR - An s-expression given to the LISP reader has faulty syntax.

BAD CHARACTER IN NUMBER - A non-numeric character was encountered while building a numeric atom. This message is also generated if an 8 or a 9 is found in an octal number.

SYMBOL TOO LONG - The printname of an atomic symbol may not be longer than 80 characters.

OVERFLO FAULT - An arithmetic operation has overflowed; it could be fixed point arithmetic overflow or an exponent over (under) flow.

FIXED OVERFLOW (TIMES) - This message is generated when a fixed point multiplication produces a product which will not fit in 36 bits. The product is truncated.

DIVIDE CHECK - This message is generated when division by zero is attempted.

1.4 Runtime Error Messages

Whenever a runtime error occurs in a LISP/66 program an error message of the following form is printed:

error message
FIRST 2 ARGS
s-expression 1
s-expression 2
TRACE BACK
trace back of stack

Most error messages are self-explanatory; however, some of them (such as CAN'T OPEN FILE) represent a large variety of errors. Appendix B is a listing of the error messages, their meaning and the significance of s-expressions 1 and 2.

Chapter 2
Built In Functions

2.1 Elementary Functions

car(x) SUBR
The function *car* returns the left half of its composite argument. Passing an atomic argument to *car* is usually an error (see *caratom*).

caratom(x) SUBR toggle
Some LISP programs (notably the LISP Compiler) need to be able to *car* through atomic symbols. Executing *caratom* with a non-NIL argument modifies *car* so that it returns a special atom if it is passed an atomic argument. This special atom prints as *FROG*, but it is not on the object list so it cannot be *eq* to anything except itself.

cdr(x) SUBR
The function *cdr* returns the right half of its composite argument. The *cdr* of an atomic symbol is the atom's property list.

caaar(x) to **cdddr**(x) SUBRS
All composite functions of *car* and *cdr* with up to three a's and d's are provided.

cons(x,y) SUBR
The function *cons* obtains a new word from free storage and builds a dotted pair of its two arguments. If the free list is exhausted, *cons* calls the garbage collector.

atom(x) SUBR predicate
atom returns T if its argument is an atomic symbol and NIL otherwise.

eq(x,y) SUBR predicate
eq returns T if its two arguments are identical list structures. *eq* should not be used to compare numbers or lists.

equal(x,y) SUBR predicate
The predicate *equal* returns T if its two arguments are the same s-expression. They do not have to have identical list structures. Fixed point and octal numbers are compared for equality, and floating point numbers are compared with a tolerance of 3.0×10**(-6). Fixed and floating point numbers may be compared with *equal*; the fixed point number is first converted to floating point.

list(x1,x2,...,xn) FSUBR
The value of *list* is a list of its arguments.

null(x) SUBR predicate
The predicate *null* returns T if its argument is NIL.

rplaca(x,y) SUBR pseudo function
rplaca replaces the left (car) pointer of its first argument with its second argument. The value of *rplaca* is x, but x has a different value than it did before the function was executed.

rplacd(x,y) SUBR pseudo function
rplacd is like *rplaca*, except that it alters the right (cdr) pointer of its first argument.

2.2 Logical Connectives

and(x1,x2,...,xn) FSUBR predicate
The arguments of *and* are evaluated in sequence from left to right, until one is found that is false, or the end of the list is reached. The value of *and* is NIL or T, respectively.

or(x1,x2,...,xn) FSUBR predicate
The arguments of *or* are evaluated in sequence from left to right, until one is found which is true, or the end of the list is reached. The return value of *or* is either T or NIL, respectfully.

not(x) SUBR predicate
The value of *not* is T if its argument is NIL, and NIL otherwise. It is the same function as *null*.

2.3 Interpreter Functions

apply(x,y,z) SUBR functional
The interpreter function *apply* evaluates the function x with arguments y using association list z.

eval(x,y) SUBR functional
The interpreter function *eval* evaluates the form x using association list y.

evlis(x,y) SUBR
The interpreter function *evlis* evaluates the elements of the list x using association list y.

evcon(x,y) SUBR
The interpreter function *evcon* evaluates the form (COND ...) The first argument is the form to be evaluated and the second argument is the association list.

function(x) FSUBR
The function *function* is used to pass functional arguments. The form (QUOTE ...) can be used instead of *function* if there are no free variables present.

2.4 Property List Functions

define(x) SUBR pseudo function
The argument of *define* is a list of pairs of the form:
((n1 l1)(n2 l2) ...(nn ln))
where each n is the name of a function and each l is the lambda expression for the function. For each pair, *define* attaches l to the property list of n using an EXPR indicator. The value of *define* is a list of the n's.

deflist(x,y) SUBR pseudo function
deflist is a more general defining function. Its first argument is a list of pairs identical to that used by *define*; its second argument is the indicator used to attach the lambda expression to the function name. *define*(x) is the same as *deflist*(x expr).

attrib(x,y) SUBR pseudo function
attrib concatenates its two arguments by changing the last element of the first argument to point to the second argument. It is useful for attaching something to the end of a property list. The value of *attrib* is the second argument.

get(x,y) SUBR
The function *get* searches the list x for an element which is *eq* to y. The value of *get* is the *car* of the rest of the list if the element is found, and NIL otherwise.

cset(x,y) SUBR pseudo function
cset is used to create a constant by attaching y to the property list of x using an APVAL indicator.

csetq(x,y) SUBR pseudo function
csetq is the *cset* except that it quotes its first argument instead of evaluating it.

put(x,y,z) SUBR pseudo function
The pseudo function *put* attaches z to the property list of x using the indicator y. *put* returns NIL.

remprop(x,y) SUBR pseudo function
remprop searches the list x, looking for all occurrences of the indicator y. When such an indicator is found, both it and the following property are removed. *remprop* returns NIL.

newname(x,y) SUBR pseudo function
newname moves the property list of x to y, replaces the property list of x with NIL, and returns y.

flag(x,y) SUBR pseudo function
flag adds the flag y to the property list of every atom in the list x. Flags are never duplicated. The value of *flag* is NIL. In LISP/66, a flag is a non-NIL property; *flag* uses the value T.

remflag(x,y) SUBR pseudo function
The pseudo function *remflag* removes all occurrences of the indicator y from the property lists of all the atomic symbols in the list x. *remflag* returns NIL.

flagp(x,y) SUBR predicate
flagp returns T if x has a non-NIL property with the indicator y; otherwise it returns NIL.

2.5 Table Building Functions

pair(x,y) SUBR
pair builds a list of pairs of corresponding elements of the lists x any y. The arguments should not be atomic symbols and must be the same length. The value of *pair* is the list of dotted pairs.

sassoc(x,y,z) SUBR functional
sassoc searches y (a list of dotted pairs) for a pair whose *car* is *eq* to x. If such a pair is found, *sassoc* returns this pair. Otherwise the value of *sassoc* is the value of function z of no arguments.

subst(x,y,z) SUBR
subst replaces all occurrences of s-expression y in s-expression z with s-expression x.

2.6 List Handling Functions

append(x,y) SUBR
append concatenates its two arguments by copying the top level of the first argument and linking the second argument to the end of this copy. The value of *append* is the resulting list.

append1(x,y) SUBR
The function *append1* is the same as
APPEND(X (CONS Y NIL)).

nconc(x,y) SUBR pseudo function
nconc concatenates its two arguments without copying the first one. The action is identical to that of *attrib* except that the value returned is the entire list (rather than the second argument).

reverse(x) SUBR
The function *reverse* reverses the top level of the list x.

length(x) SUBR
The value of *length* is the number of top level elements in the list x. Atomic symbols and () have length zero.

member(x,y) SUBR predicate
member returns T if s-expression x is *equal* to any top level element in the list y; otherwise it returns NIL.

memq(x,y) SUBR predicate
memq is like *member* except that it uses *eq* rather then *equal*.

2.7 Functionals

maplist(x,y) SUBR functional
maplist is a mapping of the list x onto a new list y(x). It is defined in LISP as:

```
(maplist (lambda (x y)
  (cond
    ((null x) nil)
    (t (cons (y x)
       (maplist (cdr x) y))) )))
```

map(x,y) SUBR functional
map is like *maplist* except that the value of *map* is NIL; *map* does not perform a *cons* of the evaluated functions. It is used when only the action of performing y is important.

mapcar(x,y) SUBR functional
mapcar is like *maplist* except that it evaluates (Y (CAR X)) instead of (Y X).

2.8 Variable Specification Functions

These pseudo functions are used to declare variables for the LISP compiler and LAP. They all return their argument.

special(x) SUBR pseudo function
The list x contains the names of variables which are to be declared *special*. The value in the *special* cell is set to NIL.

unspecial(x) SUBR pseudo function
The list x contains the names of variables which are no longer to be considered *special*.

common(x) SUBR pseudo function
The list x contains the names of variables which are to be declared *common*.

uncommon(x) SUBR pseudo function
The list x contains the names of variables which are no longer to be considered *common*.

2.9 Compiler/LAP Support Functions

bpload(x,y) SUBR pseudo function
The pseudo function *bpload* is used to store code into the binary program space, to link new SUBRs and FSUBRs into the system and to make absolute patches. The second argument of *bpload* is a list of the data to be loaded; the first argument determines how this data is to be loaded. If it is a numeric atom then its lower 18 bits are used as the base address of an absolute patch; if it is NIL then the data is stored into the binary program space.

The first argument may also be a three element list of the form (NAME IND COUNT). In this case, the data is loaded into binary program space and a standard SUBR/FSUBR link word is constructed in free space. This link word is attached to the property list of atom NAME using the indicator IND. The argument count field of the link word is set to COUNT. It is possible to memory fault the LISP system when performing absolute patches as no address checking is done.

gts(x) SUBR
The function *gts* gets the value of *special* variable x. It is an error to *gts* a variable not previously declared *special*.

pts(x,y) SUBR pseudo function
pts sets the value of *special* variable x to y and returns y. If the variable was not previously declared *special* then *pts* performs the declaration.

2.10 System Control and Debugging Functions

error(x) SUBR
Executing *error* generates a CALL TO ERROR error message and a trace back. Control is then returned to the *evalquote* listen loop.

backup() SUBR
backup is similar to *error* except that no error message or trace back is printed.

errorset(x,y) SUBR
If an error occurs during the evaluation of x, *errorset* returns NIL. The error message is printed only if y is non-NIL. If no error occurs, *errorset* returns LIST(EVAL X ALIST).

trace(x) SUBR pseudo function
The pseudo function *trace* attaches a TRACND flag to all of the function names in the list x. Whenever a function with a TRACND flag is evaluated, the system prints:

```
** TRACING
function-name  arguments
```

When the function returns, the system prints:

```
** TRACE VALUE
function-name  return-value
```

Tracing only works for EXPRs and FEXPRs; it also can produce great volumes of worthless output so it should be used with discretion.

untrace(x) SUBR pseudo function
untrace removes the TRACND flags from all of the atoms in the list x.

verbose(x) SUBR toggle
The *verbose* pseudo function controls the printing of system messages from the garbage collector. Executing *verbose* with a non-NIL argument enables the printing of the messages; executing it with a NIL argument disables the printing.

listing(x) SUBR toggle
The *listing* pseudo function enables and disables the printing of the value returned by *evalquote* in the listen loop. LISTING(NIL) is useful for supressing the printing of the value of *define* when reading a large number of functions from a disc file.

2.11 Miscellaneous Functions

save(x) SUBR pseudo function
The pseudo function *save* writes the current LISP interpreter, free storage and binary program space onto file x. This is in standard H* format, and may be system edited into timesharing or loaded with the command loader. The function then returns T. When the H* is executed control is returned to the LISP function which executed SAvE; at this point the function returns NIL.

prog2(x,y) SUBR
The value of *prog2* is y. It is used to perform two pseudo functions.

call(x) SUBR pseudo function
The printname of x is passed out to TSS via PSEUDO and CALLSS. It is necessary to use the $$ construct if the command line contains blanks (for example, CALL($$"LIST FILE44")).

gensym() SUBR
gensym creates a new atomic symbol of the form GR0000, GR0001 to GR9999. Atomic symbols created by *gensym* are not on the object list and are, therefore, unique.

genset(x) SUBR pseudo function
Executing *genset* causes the next symbol generated by *gensym* to be GRx. The argument of *genset* must be between 0 and 9999 (inclusive).

reclaim() SUBR pseudo function
reclaim causes a garbage collection and returns NIL.

peek(x) SUBR
The function *peek* is used to snap core storage. If x is a number then its lower 18 bits are used as the address to snap. If x is an alphabetic atom or a list *peek* returns a pointer to the argument in the upper half.

peek cannot memory fault the LISP system.

time() SUBR
time returns the current time of day as a two element list in hours and minutes.

proc() SUBR
proc returns the current accumulated processor time in seconds.

orderp(x,y) SUBR predicate
The function *orderp* is used to establish a canonical order among atoms. It returns T if x is ordered ahead of, or is equal to y; NIL otherwise.

2.12 System Constants

The following constants are provided in the LISP/66 system:

NAME	*PROPERTY*	*VALUE*
OBLIST	APVAL	Object List
ALIST	APVAL	Current Association List
BPSIZE	APVAL	Number of free words of binary program space
BPSORG	APVAL	Base address of free binary program space
LLENGTH*	APVAL	Output line length
	SPECIAL	Output line length
DATE*	APVAL	Date when LISP was invoked
	SPECIAL	Date when LISP was invoked

Chapter 3
Arithmetic

LISP/66 has provisions for manipulating floating point, fixed point and octal numbers.

A number is an atomic symbol and may appear in an s-expression anywhere an alphabetic symbol is legal. However, numbers are stored uniquely only on input (this is done to improve storage utilization) so they may not work properly if used as variables or function names.

3.1 Reading and Printing Numbers

Floating point numbers are distinguished by their decimal point. This decimal point cannot be the first character of the number (the reader would parse this as a LISP dot followed by a fixed point number) but it may be the last. A plus or minus sign may preceed the number, and the number may be followed by an exponent, which consists of an 'E' followed by a (signed) integer.

Spaces may be used to avoid ambiguity between a decimal point and a LISP dot; spaces are not required where no ambiguity exists.

Floating point numbers are printed in the general form sn.nnnnnnEsnn. Positive signs are never printed and the exponent is not printed if it is zero.

Fixed point numbers appear in both input and output as integers with an optional sign and exponent.

Octal numbers consist of an optional sign, up to twelve octal digits, a 'Q' and an optional octal exponent. LISP/66 handles negative octal numbers in the same manner as GMAP; the sign bit is or-ed on.

Octal numbers always print with twelve digits even though only a few digits may be significant.

3.2 Arithmetic Functions

Arithmetic functions must be given numbers as arguments; otherwise a BAD NUMBER error is generated.

Mixed mode is always permitted. Arithmetic functions will return floating point unless all of the arguments are fixed point or octal, when they return fixed point.

plus(x1,x2,...,xn) FSUBR
The value of *plus* is the sum of its arguments.

difference(x,y) SUBR
difference returns x-y.

minus(x) SUBR
The value of *minus* is -x.

times(x1,x2,...,xn) FSUBR
The value of *times* is the product of its arguments. The value of *times*() is 1.

quotient(x,y) SUBR
quotient returns x/y. If a divide check occurs the return value is meaningless.

remainder(x,y) SUBR
remainder computes the theoretic remainder for fixed point numbers and the floating point residue for floating point numbers. The return value is meaningless if a divide check occurs.

recip(x) SUBR
The value of *recip* is 1/x. The reciprocal of any fixed point number is zero.

add1(x) SUBR
add1 returns x+1. The value is fixed or floating point, depending on the argument.

sub1(x) SUBR
sub1 returns x-1. The value is fixed or floating point, depending on the argument.

fix(x) SUBR
fix converts its argument to a fixed point number.

expt(x,y) SUBR
The function *expt* evaluates x**y. If y is fixed point then repetitive multiplication is used; it if is floating point then the computation is done using logarithms and x cannot be negative.

exp(x) SUBR
The value of *exp* is e**x.

log(x) SUBR
log computes the natural logarithm of x. The argument must be positive or an error is generated.

3.3 Arithmetic Predicates

All arithmetic predicates return T or NIL.

lessp(x,y) SUBR predicate
lessp returns T if x is less than y.

greaterp(x,y) SUBR predicate
greaterp returns T if x is greater than y.

zerop(x) SUBR predicate
zerop returns T is x is zero (fixed point or octal argument) or if x < 3.0×10**(-6) (floating point argument).

minusp(x) SUBR predicate
minusp returns T is x is negative.

numberp(x) SUBR predicate
numberp returns T if x is any type of numeric atom.

fixp(x) SUBR predicate
fixp returns T if x is a fixed point number.

floatp(x) SUBR predicate
floatp returns T if x is a floating point number.

evenp(x) SUBR predicate
evenp returns T if 2 divides into x with no remainder or residue (2.0 is considered even, 2.2 is not).

3.4 Logical Operators

The logical operators perform bitwise operations on numeric atoms. They always return octal numbers.

logor(x1,x2,...,xn) FSUBR
The value of *logor* is the bitwise inclusive or of its arguments. *logor*() returns 000000000000Q.

logxor(x1,x2,...,xn) FSUBR
logxor computes the bitwise exclusive or of its arguments. *logxor*() returns 000000000000Q.

logand(x1,x2,...,xn) FSUBR
logand returns the bitwise logical and of its arguments. *logand*() returns 777777777777Q.

leftshift(x,y) SUBR
The first argument of *leftshift* is shifted by y bits. If y is positive then the shift is to the left; if it is negative the shift is to the right. All shifts are logical (zeros are shifted into unused bit positions).

3.5 Arrays

LISP programs often require the ability to manipulate indexable blocks of s-expressions. This is provided in LISP/66 by arrays.

Array pointers and array access polynomials are stored in binary program space. This space must be allocated when the LISP system is invoked (see Section 1.2).

array(x) SUBR pseudo function
array is a function of one argument, which is a list of arrays to be allocated. For example, to allocate an array A of size 7 and another array BUN of size 60 by 50, execute:

ARRAY(((A (7)) (BUN (60 50))))

ARRAY presets all of the elements of a new array to NIL. Indices range from 0 to size-1.

setel(x,y) SUBR pseudo function
The pseudo function *setel* stores s-expressions into the elements of arrays. The first argument is a subscript list of the form (array-name index1 index2 ... indexn). The second argument is the new value for the array element.

An error occurs if the specified element is beyond the limits of the array. However, no checks are made as to the number of subscripts. The last subscript of an array varies most rapidly in core. The value of *setel* is the second argument.

getel(x) SUBR
The function *getel* gets the values of array elements. The same subscripting rules given for *setel* hold for *getel*.

Chapter 4
LISP Programs

The LISP/66 program feature allows the writing of FORTRAN like programs containing LISP statements.

The PROG form has the following structure:

```
(PROG list-of-program-variables
      program-statements ...)
```

The first list after the PROG is a list of program variables. This should be written as NIL or () if there are no program variables. Variables are preset to NIL when the PROG is executed.

Program variables are set by the functions SET and SETQ. To set the program variable CRAY to 6600 execute either (SET (QUOTE CRAY) 6600) or (SETQ CRAY 6600). SETQ is usually more convenient than SET. Both SET and SETQ can also change the value of variables bound on the association list by higher level functions.

Program statements are normally executed in sequence by evaluating each one with the current association list and discarding the value. However, the function GO may be used to transfer control. Executing (GO LAB) transfers control to the label LAB (program labels are simply atomic symbols in the program body). GO can only be used inside the top level of a PROG or immediately inside a COND which is at the top level of a PROG.

Conditional expressions executed as program statements are permitted to have no true propositions. Instead of generating an error, program flow continues with the next statement.

The function *return*(x) is used to terminate a PROG. The value of the PROG is the value of x. A PROG that runs out of statements returns NIL.

Example:

rev(x) reverses a list and all of its sublists

```
    (REV (LAMBDA (X)
         (PROG (Y Z)
    A    (COND ((NULL X)(RETURN Y)))
         (SETQ Z (CAR X))
         (COND ((ATOM Z)(GO B)))
         (SETQ Z (REV Z))
    B    (SETQ Y (CONDS Z Y))
         (SETQ X (CDR X))
         (GO A) )))
```

Chapter 5
Input Output

All input output in LISP/66 is done to logical channels. There are nine disc channels (numbered 1 to 9) and one channel to the user terminal (called NIL).

5.1 Initializing Channels

The NIL channel is always initialized to the terminal. Disc channels must be initialized by the user program; this is done using the functions *openr* and *openw*.

openr(x,y) SUBR pseudo function
openr initializes channel x for input and attaches file y to it. The channel is closed if it was previously open.

If the pathname contains a slash or a dollar sign any file in the AFT with the same name is first deaccessed.

openw(x,y) SUBR pseudo function
openw initializes channel x for output and attaches file y to it. The channel is closed if it was previously open.

If the file does not exist it is created. A temporary file is created unless the pathname contains a slash or a dollar sign, when a permanent file with general read permission is created.

If the pathname contains a slash or a dollar sign any file in the AFT with the same name is first deaccessed.

close(x) SUBR pseudo function
close writes out end of file marks (output files only), releases the logical channel and deaccesses the file (if permanent and if it was brought into the AFT with *openr* or *openw*).

It is legal to *close* an inactive channel; *close* performs no action in this case.

An implicit *close* is performed on all logical channels when the user returns to the system level.

5.2 Selecting Input Output Channels

The functions *rds* (read select) and *wrs* (write select) are used to select logical channels for input and output. Both functions return the channel which was open before the input/output stream was redirected.

rds(x) SUBR pseudo function
rds causes all input to be taken from logical channel x until another *rds* is executed, or an end-of-file is encountered on the channel (when an implicit *rds*(NIL) is performed, switching input back to the terminal).

wrs(x) SUBR pseudo function
The pseudo functio *wrs* causes all output to be directed to logical channel x until another *wrs* is performed.

Disc files are automatically grown. If a request to grow is refused (input output status 17) and end-of-file is inserted into the last good block before the error message is generated.

LISP programs may be loaded from disc files by opening the file for input and selecting it:

```
? openr(1 /a/lisp/program)
1
? rds(1)
1
... program loads ...
?
```

LISP/66 switches both input and output to the terminal on errors (programs which do disc file input output under an *errorset* may be affected by this).

5.3 Input Output Functions

read() SUBR pseudo function
Executing *read* causes one s-expression to be read from the current input channel. This expression will always be read from a new line. The value or *read* is the s-expression read.

print(x) SUBR pseudo function
print writes s-expression x onto the current output unit and returns x.

prin1(x) SUBR pseudo function
prin1 writes an atomic symbol onto the current output channel without terminating the current output line. Passing a non-atomic argument to *prin1* is an

error. The value of *prin1* is x.

terpri() SUBR pseudo function
The current output line is terminated by *terpri*.

xtab(x) SUBR pseudo function
The pseudo function *xtab* writes x blanks onto the current output channel and terminates the output line if necessary. *xtab* returns x.

ttab(x) SUBR pseudo function
ttab writes enough blanks to the current output channel to make the next character print in column x. *ttab* does nothing if the output line is already past column x, and generates an error if x is greater than the current line length. *ttab* returns x.

otll(x) SUBR pseudo function
The current line length is set to x by *otll*, which returns its argument. The new line length must be between 40 and 120 inclusive.

prompt(x) SUBR pseudo function
The function *prompt* changes the terminal input prompt to the printname of x. The new prompt must be four characters or less in length.

If x is NIL then the prompt is turned off completely (changing the prompt to NIL is impossible).

Chapter 6
Character Manipulation

Characters in LISP/66 are ordinary atomic symbols with single character printnames; the alphabetic atom A and the character A are identical.

Since characters are on the object list they may be compared using *eq*. However, for compatibility with other LISP systems, using *eq* is not recommended; using *equal* (LISP 1.6) or *cclass* (LISP/360) is a better practice.

6.1 Character Input Output

princ SUBR pseudo function
princ writes the character x onto the current output channel and returns x. *princ* is the same function as *prin1*.

readch() SUBR pseudo function
The function *readch* reads and returns the next character from the current input channel. Lower case characters are mapped into upper case.

endread() SUBR pseudo function
The execution of *endread* forces the next *readch* to a new line. It is commonly used to skip over the remainder of an input line when an error is detected.

passcr(x) SUBR pseudo function
Executing *passcr* with a non-NIL argument causes *readch* to begin passing carriage returns to the user program. This mode is disabled by *passcr*(NIL). *passcr*(NIL) is the default.

6.2 Character Functions

explode(x) SUBR
explode takes its argument (which must be an atomic symbol) and returns a list of its constitutent characters. *explode* works for all types of atoms, including floating point.

compress(x) SUBR
compress takes a list of characters and *compress*ed them into an atomic symbol.

In order to decide what type of atom to construct, *compress* skips over leading plus and minus signs and examines the next character. If this character is a digit then *compress* builds a number; otherwise it builds an alphabetic atom. It is impossible to build an alphabetic atom with a printname like 8888 using *compress*.

6.3 Character Predicates

liter(x) SUBR predicate
liter returns T if its argument is a letter (between A and Z).

ditgit(x) SUBR predicate
digit returns T if its argument is a *digit* (between 0 and 9).

cclass(x,y) SUBR predicate
cclass is a general character predicate. It returns T if the character x is in the printname of y.

Chapter 7
Internal Formats

This chapter is for general (why did my strange function memory fault and blow me to system level?) information only; more detailed descriptions of LISP/66 internal formats may be found in the LISP/66 SYSTEM MAINTENANCE MANUAL (whenever it is written).

7.1 LISP Cells

A LISP cell occupies one 36 bit machine word in the following format:

Bit 0	Must be zero
Bits 1-17	Car pointer
Bit 18	Usually zero. Used by garbage collector
Bits 19-35	Cdr pointer

Since 17 bit addresses are used LISP/66 can only handle 128K of free storage. This should cause no problems as TSS EXEC aborts programs larger than 80K when it attempts to swap them.

7.2 Atomic Symbols and Property Lists

An atomhead is a LISP cell with bit 0 equal to 1. The car pointer of the atomhead contains the atom's type; 0 for alphabetic atoms, 1 for octal numbers, 2 for fixed point numbers and 3 for floating point numbers.

The cdr pointer points to a word which has a pointer to the atom's printname in bits 0-17 and a pointer to the property list in bits 18-35.

Printnames are stored as forward linked lists with two characters in the upper half and a link pointer or NIL in the lower half. Short printnames are padded with nulls.

The value of a numeric atom is stored in two halves (like an alphabetic printname) to simplify garbage collection; the first word in the list contains bits 0-17 of the number.

Property lists have the same structure as those in LISP.1.5.

7.3 SUBRs and FSUBRs

The SUBR and FSUBR link word is attached to the property list of the function name using a SUBR or FSUBR indicator. The link word has the number of arguments in the upper half (FSUBRs have zeros in the upper half) and the pointer to the routine in the lower half.

SUBRs and FSUBRs are called by a TSX1 and return with a TRA 0,1.

7.4 Sample Atom

```
*       The atom BPSIZE
        VFD     O18/400000,18/*+1
        ZERO    *+1,*+4
        VFD     O18/102120,18/*+1
        VFD     O18/123111,18/*+1
        VFD     O18/132105,18/NIL
        ZERO    APVAL,*+1
        ZERO    *+1,NIL
        ZERO    *+1,NIL
        VFD     O18/400001,18/*+1
        ZERO    *+1,NIL
        ZERO    0,*+1
BPSIZE  ZERO    0,NIL
```

Appendix A
Known Problems

RETURN and GO do not work as arguments of a PROG2 (this is destined to stay around for a long time).

The constructs *eval*(x,y) and *apply*(x,y,z), where x is not bound on the property list and y is a non-NIL atom, cause the system to memory fault.

Appendix B
Error Messages

TOO FEW ARGS (SUBR)
TOO MANY ARGS (SUBR)
The wrong number of arguments have been passed to a LISP function.
S-expression 1 is the function.
S-expression 2 is the list of arguments.

UNDEFINED FUNCTION (APPLY)
UNDEFINED FUNCTION (EVAL)
An atom has been used as a function but has never been defined.
S-expression 1 is the function.
S-expression 2 is the association list.

UNBOUND VARIABLE
The variable is not defined as a function argument on the association list and does not have an assigned value.
S-expression 1 is the unbound variable
S-expression 2 is the association list.

TOO MANY AGRS
LISP/66 cannot pass more than 20 arguments to a function.
S-expression 1 is the list of arguments.
S-expression 2 is not valid information.

UNSATISFIED COND
No true propositions were found in a COND.
S-expressions 1 and 2 are the arguments to *evcon*.

CAR OF ATOM
An atomic symbol has been passed to *car*.
S-expressions 1 and 2 are the argument of *car*.

BAD ADDRESS
peek has been passed a list or an address beyond LISP/66's address limits.
S-expression 1 is the argument of *peek*.
S-expression 2 is not valid information.

BAD ARGUMENT
A LISP function has been passed an argument which is not compatible with the function.
S-expression 1 is the bad argument.
S-expression 2 is not valid information.

BAD NUMBER
An arithmetic function has been passed a non-numeric argument.
S-expression 1 is the argument.
S-expression 2 is not valid information.

SET VAR UNDEF
The function *set* or *setq* has been given an undefined program variable.
S-expression 1 is the program variable.
S-expression 2 is the association list.

NON ATOMIC ARG (PRIN1)
The argument of *prin1* is a list.
S-expression 1 is the argument.
S-expression 2 is not valid information.

GO LABEL UNDEF
The label given as the argument of *go* has never been defined.
S-expression 1 is the undefined label.
S-expression 2 is the golist (list of all labels).

TOO MANY ARGS (EXPR)
TOO FEW ARGS (EXPR)
The wrong number of arguments have been passed to a defined function.
S-expression 1 is a list of the function variables.
S-expression 2 is the list of supplied arguments.

BAD CHARACTER
The argument passed to a character function or predicate is not a valid character atom.
S-expression 1 is the character argument.
S-expression 2 is not valid information.

BAD COMPRESS
The list of characters passed to *compress* could not be made into a legal atom.
S-expressions 1 and 2 are not valid information.

BAD SAVE
The LISP core image could not be written out successfully.
S-expression 1 is the file name.
S-expression 2 is not valid information

OUT OF BINARY PROGRAM SPACE
There is not enough binary program space left to execute a function. This message is generated by *bpload* and *array*.
S-expressions 1 and 2 are not valid information.

NO MORE CORE
TSS has refused a request to obtain more free space.
S-expressions 1 and 2 are not valid information.

CALL TO ERROR
The function *error* has been called.
S-expression 1 is the argument of *error*.
S-expression 2 is not valid information.

BAD MEDIA ON INPUT
The currently selected input file is not media 6 ASCII.
S-expressions 1 and 2 are not valid information.

WRS ON INPUT FILE
The logical channel selected for output has been opened for input.
S-expression 1 is the logical channel number.
S-expression 2 is not valid information.

FILE AT EOF
The selected input channel is positioned at end-of-file.
S-expression 1 is the logical channel number.
S-expression 2 is not valid information.

RDS ON OUTPUT FILE
The logical channel selected for input has been opened for output.
S-expression 1 is the logical channel number.
S-expression 2 is not valid information.

FILE NOT OPEN
The logical channel given as an argument to *rds* or *wrs* has never been opened.
S-expression 1 is the logical channel number
S-expression 2 is not valid information.

BAD CALL (.LN)
The logarithm routine has been passed a negative argument. This message is generated by *log* and *expt*.
S-expression 1 is the argument.
S-expression 2 is not valid information.

BAD CALL (.EXP)
The exponentiation routine has been passed an argument greater than 88.5. This message is generated by *exp* and *expt*.
S-expression 1 is the argument.
S-expression 2 is not valid information.

GTS VAR UNDEF
The argument of *gts* was never declared *special*.
S-expression 1 is the argument of *gts*.
S-expression 2 is not valid information.

FATAL ERROR: PDS OVERFLOW IN GCL
The pushdown stack has overflown during garbage collection. LISP/66 has terminated.

SUBSCRIPT ERROR
The subscript list specifies an array element beyond the limits of the array. This message is generated by *setel* and *getel*.
S-expression 1 is the subscript list.
S-expression 2 is the new value (*setel*)
S-expression 2 is not valid information (*getel*)

CAN'T OPEN FILE
This message is generated for any *openr*/*openw* file system error, such as syntax error in the pathname, permissions denied, file does not exist (*openr*) or AFT full.
S-expressions 1 and 2 are the arguments of the *openr* or *openw*.

CAN'T CLOSE FILE
A disc file will not close properly (usually an IOS status 17 on the last block).
S-expression 1 is the logical channel number.
S-expression 2 is not valid information.

STATUS 17, CHANNEL x
The disc file on channel x will not grow.
S-expressions 1 and 2 are not valid information.

Appendix C
Glossary

association list	A list of pairs of terms which is equivalent to a table with two columns. It is used to pair bound variables with their values.
atom	A synonym for atomic symbol.
atom symbol	The basic constituent of an S-expression.
bound variable	A variable included in the list of bound variables after a LAMBDA is bound within the scope of the LAMBDA. This means that its value is the argument corresponding in position to the occurrence of the variable in the LAMBDA list.
free-storage list	The list of free words in the computer memory. Each time a *cons* is performed the first word of the free-storage list is removed. When the free-storage list is exhausted, a new one is built by the garbage collector.
free variable	A variable that is neither a program variable or a bound variable.
functional	A function that can have functions as arguments. *apply*, *eval*, *sassoc* and the mapping functions are functionals in LISP/66.
functional argument	A function that is an argument for a functional. In LISP, a functional argument is quoted by using the special form (FUNCTION fn).
garbage collector	The routine in LISP/66 which identifies all active list structure by tracing it from fixed base cells and marking it, and then collects all unneeded cells (garbage) into a new free-storage list.
GMAP	General Macro Assembly Program the assembler for the HIS 6000.
indicator	An atomic symbol occurring on a property list that specifies that the next item on the list is a certain property. EXPR, SUBR, FEXPR, FSUBR and APVAL are examples of indicators.
interpreter	An interpreter executes a source language program by examining the source language and performing the specified algorithms. This is in contrast to a compiler which translates a source language program into machine language for subsequent execution. LISP/66 is an interpreter.
predicate	A function whose value is true or false. In LISP/66 false is represented by NIL and true by anything that is non NIL.
program variable	A variable that is declared in the list of variables following the word PROG. Program variables have initially the value NIL, but can be assigned other values by *set* and *setq*.

property	An expression associated with an atomic symbol.
property list	The list of an atom's properties; the CDR of an atom is the atom's property list.
pseudo function	A function which has effects other than delivering a value. For example, *read* or *rplaca*.
recursion	The technique of defining a function in terms of itself.