APPENDIX C: LISP/360 REFERENCE MANUAL

Revised February 1968

Campus Facility USERS MANUAL

PREFACE

This paper is intended to provide the LISP 1.5 user with a reference manual for the LISP 1.5 interpreter on the Campus Facility 360/67. This manual assumes that the reader has a working knowledge of LISP 1.5 as implemented on the IBM 7090. Care has been taken to ensure compatibility between LISP/360 and LISP 1.5. For those new to LISP 1.5 or for those who feel a need for a refresher, the LISP 1.5 Programmer's Manual [1] is suggested.

The particular implementation to which this reference manual is directed was started by Mr. J. Kent while he was at the University of Waterloo [4,5]. It is modeled after his implementation of LISP 1.5 for the CDC 3600 [4].

Since the last edition of this manual, Messrs. Kent and Berns have completed the LISP 1.5 Assembler and Compiler. Information on these processors is included in this paper.

Rod Fredrickson
Associate Director,
Campus Facility
Stanford Computation Center

TABLE OF CONTENTS

1. THE LISP/360 SYSTEM

LISP/360 operates under the IBM System/360 Operating System. The actual implementation of the system differs in some important respects from the IBM 7090 LISP 1.5. These changes were instituted to increase the efficiency of the system. The most marked differences are in the organization of storage, where the idea of a separate block for 'fullword-storage' has been abandoned, and in the organization of the internal representation of LISP-atoms. Several of the indicators needed on the property lists in LISP 1.5 have been rendered unnecessary. Also, the interpreter has been assembled relocatable.

1.1 Organization of the System

1.1.1 Organization of Storage

The interpreter, the stack, and freecellstorage in LISP/360 are all contained in one control section. Many versions of the system may be prepared with the following recommended requirements.

VERSION	STACK SIZE	NO. OF LISP CELLS	MEMORY REQUIREMENTS
LISPA	4000	20,500	200K
LISPB	6000	40,000	400K
LISPC	8000	65,000	600K

The current Stanford implementation is one system called LISPA with 6000 stack elements and 40,000 free cells (when the Binary Programming Space for the compiler is replaced with free cells via the function bpsz -- otherwise it is 30,000).

A LISP-cell is a double word (64 bits) in LISP/360. A stack unit is a single word (32 bits). The size of the stack and freecellstorage can only be changed by reassembling the LISP/360 system.

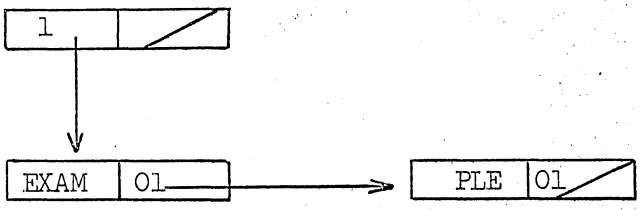
1.1.2 Object List

That part of the object list which contains the standard atoms has been generated in assembly language. The object list is not bucket sorted as in LISP 1.5.

1.1.3 Atoms

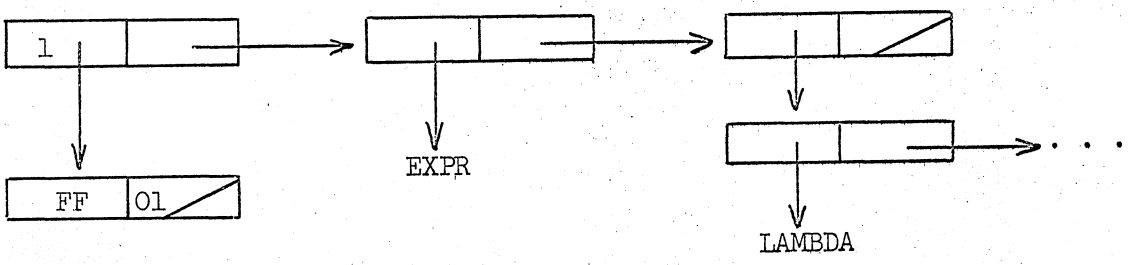
The atoms and their property lists have been reorganized in LISP/360. All LISP-cells having bit 0 set are so-called atomheads. An atomhead contains in its upper address a pointer to the atom's fullcell list and in the lower address a pointer to the atom's property list.

The atom EXAMPLE with an empty property list:



1.1.4 Property Lists

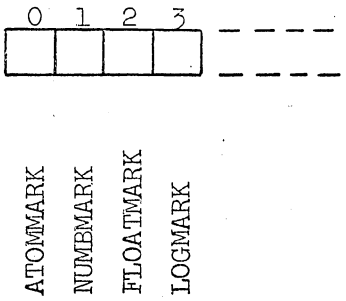
A typical property list might look like this:



FF is a function, namely an EXPR which starts this way (LAMBDA (X)...).

1.1.5 Binary Markers

Since the LISP-cell length is 64 bits and only 24 bits are needed to express an address, 8 bits in the upper word and 8 bits in the lower word are released for other uses. As mentioned above, bit 0 indicates that the cell is an atomhead.



0	1	2	3	
1	0	0	0	BCD ALFAMER
1	1	0	0	FIXPOINT
1	1	1	0	FLOATPOINT
1	1	0	1	LOGICAL

The bits 1, 2, and 3 refer to the fullcell list associated with the atom.

A function that is to be traced has bit 7 set. This means that the indicator TRACE is not needed.

When bit 33 is set, this indicates that the word in question is a fullcell. Bit 32 is used by the garbage collector to mark active cells.

Bit 34 is now set in a fullcell when its upper 32 bits contain BCD characters or a number. Bit 34 is not set in a fullcell when its upper 32 bits is an address. This relocation marker is used in chkpoint to determine what fullcells should have its car parts made relocatable.

1.1.6 Fullcells

The fullcells in free cell storage replace the 'fullwordstorage' in LISP 1.5.

A fullcell is a LISP-cell with bit 33 set and the upper 32 bits occupied by either:

a. Four BCD characters from a printname (if need be, filled in from the right with zeros), or

b. a 32 bit number, or

c. the address of a binary LISP-routine (SUBR or FSUBR).

1.1.7 Printnames

All non-numeric atoms have in the upper address of their atomhead the address of a linear list containing their BCD printnames. For instance, the atom DIFFERENCE has this fullcell list:

1.1.8 Numbers

There are three kinds of numbers:

a. Fixed-point (integers)

b. Floating-point

c. Logical

All numbers are stored as 32 bit binary numbers with the help of a fullcell, and must be converted to BCD on input and output. (The BCD representation of a number is not stored.)

The correct form of a logical or hexadecimal number is as follows:

1) The number must start with a decimal digit.

2) The number should terminate with the letter X.

3) A scale factor may follow the X. The scale factor must be a decimal integer, no sign allowed. The number is shifted left 4 bits times the scale factor. Thus the scale factor is an exponent to the base 16.

Examples:

External form	Internal form
0FFX6	FF000000
0FFFFFFFFX	FFFFFFFF
9A6X2	0009A600

2. IMPLEMENTED FUNCTIONS

The functions that are marked by an asterisk are new functions or functions that are different from the functions with the same name in IBM 7090 LISP 1.5. See Section 2.1 for details on these differences. Information about the other functions can be found in McCarthy [1].

addl[n]

and[x_1;x_2;...;x_n]

append[x;y]

* appendl[x;y]

apply[fn;args;a]

atom[x]

attrib[x;y]

breakp[ch]

car[x]

caar[x]

caaar[x]

caadr[x]

cadar[x]

caddr[x]

cadr[x]

cdaar[x]

cdadr[x]

cdar[x]

cddar[x]

cdddr[x]

cddr[x]

cdr[x]

cond[$p_1 \rightarrow e_1$;$p_2 \rightarrow e_2$;...;$p_k \rightarrow e_k$]

cons[x;y]

cset[x;y]

csetq[x;y]

define[x]

deflist[x;ind]

difference[x;y]

* digp[ch]

eject[ ]

eq[x;y]

equal[x;y]

error[x]

eval[form;a]

* evenp[n]

evcon[x;a]

evlis[x;a]

* explode[x]

* expt[n;m]

* fix[n]

fixp[n]

flag[x;flag]

flagp[x;flag]

* float[n]

floatp[n]

function[x]

* gensyml[x]

get[x;ind]

go[label]

greaterp[x;y]

label[name;fn]

* last[x]

leftshift[m;n]

length[x]

* litp[ch]

lessp[x;y]

list[x_1;x_2;...;x_k]

logand[n_1;n_2;...;n_k]

logor[n_1;n_2;...;n_k]

* logp[n]

logxor[n_1;n_2;...;n_k]

mapcar[x;fn]

maplist[x;fn]

max[n_1;n_2;...;n_k]

member[x;y]

min[n_1;n_2;...;n_k]

minus[n]

minusp[n]

* mkatom[ ]

nconc[x;y]

not[x]

null[x]

numberp[x]

or[x_1;x_2;...;x_k]

* orderp[x;y]

pair[x;y]

plus[n_1;n_2;...;n_k]

* prbuffer[x]

prin1[x]

print[x]

prog[varlist;statlist]

prog2[x;y]

quote[x]

quotient[x;y]

read[ ]

* readch[x]

recip[n]

reclaim[ ]

remainder[n;m]

remflag[x;flag]

remob[x]

remprop[x;ind]

* rlit[ch]

* rnumb[ch]

rplaca[x;y]

rplacd[x;y]

* rtoken[fn]

sassoc[x;a;fn]

set[x;y]

setq[x;y]

subl[n]

terpri[ ]

times[n_1;n_2;...;n_k]

trace[x]

* ttab[n]

untrace[x]

verbos[x]

* xtab[n]

zerop[n]

2.1 New Functions

appendl[x;y] = nconc[x;cons[y;NIL]]

breakp[x] is a predicate. If its argument is one of these character-objects: blank ( ) , .
its value is T; otherwise its value is F.

digp[ch] is a predicate. If its argument is one of these character-objects: 0, 1, 2, ... , 9
its value is T; otherwise its value is F.

evenp[n] takes an integer as an argument and returns the value T if the number is even, otherwise F.

explode[x] takes an atom as an argument and has as its value a list of the characters in the atom's printname.

expt[n;m] computes n^m. expt will only accept an integer exponent. The value of expt when the exponent is negative is a floating-point number, e.g., expt[2;-1] = 0.5.

fix[n] will convert a floating-point number into a fixed-point number.

float[n] will convert a fixed-point number into a floating-point number.

gensym1[x] generates new, distinct atomic symbols. Its argument should be an atom and the first four characters in this atom's printname will be used as the four first characters in the new atom's printname. For example:

gensym1[ALPHA] = ALPH0001

last[x] = [null[cdr[x]→ x;last[cdr[x]]]]

letp[x] is a predicate. If its argument is one of these letters A, B, ... , Z its value is T; otherwise its value is F.

litp[ch] = not[or[breakp[x];digp[x]]]

logp[x] is a predicate with the value T if its argument is a logical number, otherwise F.

mkatom[ ] This function with no arguments is used to make atoms out of information put into the internal character buffer by rlit or rnumb.

orderp[x;y] induces an arbitrary canonical order among atomic symbols.

prbuffer[x] takes T or NIL as an argument. prbuffer[T] will cause read and readch to print the input buffer everytime a new card is moved into it. A => in the margin of a line indicates that the line is a buffer printout. prbuffer[NIL] will stop the printing of the input buffer. prbuffer is used when it is important to show exactly what was given as input to LISP.

readch[x] takes T or NIL as arguments. If the argument is NIL, readch will read the next character from the input buffer and return with the corresponding character-object as a value. readch[T] causes a simulated backspace. The value of readch[NIL] after a readch[T] has been executed will be the same as that returned by the previous readch[NIL]. The value of readch[T] is the same as that returned by the next to last readch[NIL]. readch[T] should only be executed once before calling readch[NIL].

rlit[x] takes a character-object as an argument and puts the corresponding character into an internal character buffer. Executing rlit sequentially will cause a string of characters to be constructed in the character buffer. mkatom can then be called to make a literal atom out of it.

rnumb[x] takes one of these character-objects as an argument:

+
-
.
E
0, 1, 2, ..., 9.

rnumb will construct a partially translated number in the internal character buffer. Remember that the character-objects 0, 1, 2, ..., 9 are different from the numbers 0, 1, 2, ..., 9. The sequence of character-objects presented to rnumb, one at a time, must represent a meaningful integer or floating-point number. mkatom can then be called to make a numeric atom out of the information in the character buffer.

rtoken[fn] As can be seen from the following definition rtoken takes a functional argument. The function supplied as an argument should have character-objects as values, and should have the same backspace facilities as readch has. Depending on the character-objects supplied by rtoken's functional argument, rtoken will give a literal atom, a numeric atom, or one of the character-objects -- blank + - , . ( ) -- as a value.

```
rtoken[fn]=prog[[ch;cht];
      A   ch:=fn[NIL];
          [eq[ch;BLANK]→go[A]];
          [or[eq[ch;-];eq[ch;+]]→go[NAS]];
          [eq[ch;$]→go[$1];
          litp[ch]→go[LA];
          digp[ch]→go[NA];
          breakp[ch]→return[ch]];
```

```
LA    rlit[ch];
LC    ch:=fn[NIL];
      [breakp[ch]→ prog2[fn[T];go[LB]]];
      go[LA];
LB    return[mkatom];
NA    rnumb[ch];
      ch:=fn[NIL];
      [and[breakp[ch];not[eq[ch;.]]]→ go[LB]];
      go[NA];
NAS   [breakp[fn[NIL]]→ prog2[rlit[ch];go[LB]]];
      ch:=fn[T];
      go[NA];
$1    ch:=fn[NIL];
      [eq[ch;$]→ go[$2]];
      rlit[$];
      fn[T];
      go[LC];
$2    cht:=fn[NIL];
$3    ch:=fn[NIL];
      [eq[ch;cht]→ return[mkatom]];
      rlit[ch];
      go[$3]];
```

ttab[n] will set the first n positions in the output buffer to blanks.

xtab[n] will insert n blanks in the line currently being built up in the output buffer.

xtab and ttab will only affect one line at a time. ttab must be repeated for each line outputted to get a margin on the page.

2.2 New Data Management Functions

```
asa[x]
bpsz[n]
chkpoint[ddname]
close[ddname]
```

inll[n]

letp[x]

open[ddname;dcbdesc;iospec]

otll[n]

rds[ddname]

restore[ddname]

wrs[ddname]

A detailed explanation of these functions can be found in Section 6.

2.3 Atoms with Initial Values

Several atoms have predefined values (APVALS) in LISP/360. These atoms and their corresponding values are:

Atom	Value
NIL	NIL
F	NIL
T	T
OBLIST	Object list
ALIST	Association list
DOLLAR	$
SLASH	/
LPAR	(
RPAR	)
COMMA	,
PERIOD	.
PLUSS	+
DASH	-
STAR	*
BLANK	blank
EQSIGN	=

2.4 Character-objects

The following character-objects are defined in the system.

blank	(	!	X	4
A	+	$	Y	5
B	|	*	Z	6
C	&	)	unprintable	7
D	J	;	,	8
E	K	¬	%	9
F	L	-	_	:
G	M	/	>	#
H	N	S	?	@
I	Ø	T	0	'
¢	P	U	1	=
.	Q	V	2	"
<	R	W	3	

The 'unprintable' character has no graphic symbol on the printer. Its punched card code is 12-11. readch will translate any one of the 256 characters available on the IBM System/360 into one of the above-mentioned 64 character-objects. Small letters are translated into capital letters.

3. LISP/360 PECULIARITIES

1. Alphameric atoms in LISP/360 may have up to 80 characters.
2. Fixed-point numbers may have absolute values up to 2^{31}.
3. Floating-point significance on input is 6 digits.
4. Floating-point numbers may have absolute values between 10^{75} and 10^{-75} (and 0).
5. Numbers are considered equal if the absolute value of their difference is less than 10^{-6}.
6. CAR of an atom is not junk as in LISP 1.5, but the address of the fullcell list of that atom.
7. No control cards of any type exist in LISP/360.
8. Signs are ignored in reading logical numbers.
9. If a print is executed after prinl, the list generated by print follows the data output by prinl.
10. go can only be given atomic labels.

4. OPERATING PROCEDURES

4.1 Running a Program Punched on Cards

The LISP program can be punched on cards free field in columns 1-72. The following control cards are necessary to run the LISP program from cards:

```
<JØB Card>
//JØBLIB  DD  DSNAME=SYS2.PRØGLIB,DISP=ØLD
//stepname  EXEC  PGM=LISPA
//LISPØUT  DD  SYSØUT=A
//LISPIN  DD  *
    LISP Program
/*
```

A LISP program that has been punched on an IBM 026 keypunch (BCD) can still be interpreted by this interpreter by introducing the parameter BCD in the EXEC statement. The EXEC statement would then be

```
//stepname  EXEC  PGM=LISPA,PARM=BCD
```

( ) + are the only characters translated from BCD to EBCDIC by the interpreter when the BCD parameter is included in the EXEC card. The character = is represented differently in BCD and EBCDIC and is not translated by the above-mentioned technique.

No translation is performed by the readch function even if the BCD parameter is included in the EXEC statement.

5. LISP/360 SYSTEM MESSAGES

5.1 evalquote Messages

The message ARGUMENTS FOR EVALQUOTE... and the two S-expressions in the last doublet are always printed before entering evalquote.

If no errors occur during the evaluation of the doublet, the message TIME xxxxMS, VALUE IS... and the value of evalquote for this doublet is printed upon return from evalquote. The time indicated in the above message gives the time spent in evalquote. The time is in milliseconds.

5.2 Tracing in LISP/360

Tracing is controlled by the pseudo-function trace, whose argument is a list of functions to be traced. After trace has been executed, tracing will occur whenever these functions are entered. The trace-handler prints out the name of a function and a list of its arguments when it is entered, and its name and value when it is finished. When tracing of certain functions is no longer desired, it can be terminated by the pseudo-function untrace whose argument is a list of functions that are no longer to be traced.

5.3 Garbage Collector Message

The message COLLECTED xxxxx CELLS AND STACK HAS xxxx UNITS LEFT is printed after every garbage collection. The message gives an indication of the amount of freecellstorage freed, and the depth of recursion at each garbage collection. The system parameters are defined so that there are 4000 units in the stack and 20000 LISP cells available for programmer use in freecellstorage in LISPA.

5.4 Interruption Message

An interrupt supervisor takes care of all program interruptions in LISP/360. See the IBM reference [3] for information about System/360 interruptions. The program status word (PSW), the contents of registers 1-16 and the message ***ERROR: CAR TAKEN OF FULLCELL is printed if the interruption code is 1 to 7. Thereafter a trace back follows of the same type as described in Section 5.5.2. This interruption type is

usually caused by indiscriminate use of car and cdr past the atomic level. The execution of the doublet that caused the interruption is halted and a new doublet is read in for evaluation. An interruption code of 8 to F means that an overflow or underflow occurred. This interruption type causes the message ***OVER-OR UNDERFLOW OF TYPE xx to be printed. xx is the interruption code. Execution of the doublet that caused the overflow or underflow is resumed after the interruption.

5.5 Error Diagnostics

5.5.1 Syntax Errors

If the reader finds syntactical errors in an S-expression, it inserts special atoms at appropriate places in the S-expression. The special atoms have the following meaning.

ATOM	MEANING
ERRB	. (dot) encountered as first non-blank character after a (.
DOTERR1	The second S-expression in a dotted pair is not followed by a right parenthesis.
DOTERR2	A . or) encountered as first non-blank character after a dot.

The message ***R1-SYNTAX ERROR precedes the printing of the S-expression with the error. A doublet containing one or more syntactical errors causes the following message to appear ***ERRORS ENCOUNTERED WHILE READING. CONTINUING WITH NEXT DOUBLET and evaluation of the doublet is skipped.

5.5.2 Runtime Errors

When an error occurs during a LISP run, the following type of error diagnostic occurs.

```
***error code-error message
   S-expression 1
   S-expression 2
***TRACE BACK FOLLOWS
   S-expression 3
      :
      :
```

S-expressions 1 and 2 are related to the type of error encountered and are described below with the error messages. The trace back is a printing of the lists bound on the stack at the time the error occurred. The most recently used list in the stack (the list on top) is printed first. The first few printed lists will therefore give a good indication of what caused the error.

Let us assume that none of the functions being interpreted are using the prog-feature, and that trace has not been executed. Under these conditions, the lists bound on the stack will be alternately function calls or definitions and association lists. When reading the stack, keep in mind that the innermost functions are evaluated first, even though the functions are interpreted from the outside in. Thus, the call on the function being evaluated when the error occurred will be near the top of the stack.

If trace is executed within a LISP job, the name of an EXPR called will be found on the stack between the EXPR's definition and the corresponding association list. The call on a function using the prog-feature will cause the following lists to appear in the stack printout.

a. The complete function definition (omitting the name of the function).
b. A list of the uninterpreted statements in the function starting with the one being evaluated when the error occurred.
c. The go-list (see reference [4]).
d. The association list.

5.5.3 Error Codes and Messages

A1-CALL TO ERROR

This message is given if a LISP program calls error. The argument (if any) of error is printed (S-expression 1). The trace back is not given with this message.

A2-FUNCTION NOT DEFINED

This message occurs when an atom given as the first argument of apply does not have a function definition either on its property list or on the association list.

S-expression 1 is the atom in question.

S-expression 2 is the association list.

A3-NO ARGS OF COND TRUE

None of the propositions following cond are true.

S-expression 1 is the list of the arguments given cond.

S-expression 2 is the association list.

A5-SET VARIABLE UNDEF

The function set or setq was given an undefined program variable.

S-expression 1 is the program variable.

S-expression 2 is the association list.

A6-UNDEF LABEL IN GO

The label given as the argument of go has not been defined.

S-expression 1 is the label.

S-expression 2 is the list of the labeled statements.

A7-MORE THAN 22 ARGS

More than 22 arguments given to an EXPR or a SUBR.

S-expression 1 is the list of arguments to the function.

A8-UNDEFINED VARIABLE

A variable is not bound on the association list, nor does it have an APVAL. This error occurs in eval.

S-expression 1 is the variable in question.

S-expression 2 is the association list.

A9-FUNCTION NOT DEFINED

The form given as the first argument to eval has as its first element an atom with no function definition either on its property list or on the association list.

S-expression 1 is the atom in question.

S-expression 2 is the association list.

D2-FILE CANNOT BE OPENED - NO STORAGE AVLBL

open was asked to open a data set (file) when there was no storage available in which to put the DCB for that data set. close releases the space taken up by the DCB of the data set that it is closing.

S-expression 1 is the ddname given as the first argument to open.

D3-RDS FILE NOT OPENED

D4-WRS FILE NOT OPENED

A data set (file) must be opened by open before LISP/360 can write or read from it.

S-expression 1 is the ddname given as the argument to rds or wrs.

D5-CHKPOINT FILE NOT OPENED

D6-RESTORE FILE NOT OPENED

A data set (file) must be opened by open before chkpoint or restore can use it.

S-expression 1 is the ddname given as the argument to chkpoint or restore.

D7-RESTORE GIVEN FILE INCOMPATIBLE WITH SYSTEM SPECIFIED

A data set (file) produced by chkpoint under LISPA cannot be restored under LISPB or LISPC, or vice versa.

F2-TOO MANY ARGUMENTS-EXPR

F3-TOO FEW ARGUMENTS-EXPR

The wrong number of arguments has been given to a defined function.

S-expression 1 is the list of the function variables.

S-expression 2 is the list of supplied arguments.

F2-TOO MANY ARGUMENTS-SUBR

F3-TOO FEW ARGUMENTS-SUBR

The wrong number of arguments has been given to a SUBR.

S-expression 1 is the function.

S-expression 2 is the list of arguments.

G2-PUSHDOWN STACK OVERFLOW

Recursion is very deep. Non-terminating recursion will cause this error. S-expression 1 and 2 will, if given, depend on where in the interpreter the stack was last used. The traceback is not given on this error. The message IN THE GARBAGECOLLECTOR may follow immediately after this message. - This means that there was not enough stack left for the garbage collector to work with, when the garbage collector was called. This is a fatal error, and LISP/360 gives up control to OS/360.

GC2-STORAGE EXHAUSTED

The garbage collector is unable to find any unused cells in free-cellstorage. S-expression 1 and 2 are the arguments of cons. The traceback is not given on this error. This is a fatal error, and LISP/360 gives up control to OS/360.

I3-BAD ARITHMETIC ARGUMENT

An arithmetic routine was given a non-arithmetic argument. S-expression 1 and 2 will depend on which arithmetic routine found the error.

I5-ATTEMPT TO RAISE 0 TO 0

This error is caused by trying to execute either expt[0;0] or expt[0.0;0].

I6-ATTEMPT TO RAISE 0 TO NEGATIVE POWER

This error is caused by trying to execute either expt[0;n] or expt[0.0;n], where n is negative.

I8-EXPT CANNOT TAKE REAL EXPONENT

This error occurs when the second argument of expt is a floating-point number.

R1-SYNTAX ERROR

A syntax error has occurred while reading an S-expression. S-expression 1 is the S-expression in question. The traceback is not given on this error.

R2-BAD BRACKET COUNT

An end-of-file was reached while reading an S-expression. S-expression 1 is the list as read with needed brackets generated. The traceback is not given on this error. This is a fatal error and LISP/360 gives up control to OS/360.

R3-BAD BRACKET COUNT ON USER FILE

An end-of-file was reached while reading an S-expression from a data set other than LISPIN. S-expression 1 is the list as read with needed brackets generated. The traceback is not given on this error. The error causes LISP to start reading from LISPIN.

R5-NAME OR NUMBER TOO LONG

A BCD printname or a number is longer than that accepted by the interpreter. Truncation occurs on the right. Only the message appears with this error.

6. DATA MANAGEMENT IN LISP/360

6.1 Data Management Functions

LISP/360 can read or write data sets on any OS/360 supported device with the aid of the functions open, close, wrs, and rds. LISP's handling of its buffers can be modified by the functions asa, inll, and otll. It is assumed in the following that the reader has a working knowledge of OS/360 Data Management.

6.1.1 open[ddname;dcbdesc;iospec]

All data sets must be 'opened' by the function open before they are used. A DD card is used to define the data set and open uses the ddname on the card to refer to the data set. The ddname is the first arguemnt of open. The record length (LRECL), the blocksize (BLKSIZE), and whether or not the record's first character is a control character (A), can be specified in the second argument of open. The third argument of open specifies whether the data set is to be used for input (INPUT) or output (OUTPUT).

An example of the opening of the data set defined on the DD card named DATA:

```
OPEN(DATA ((LRECL.100)(BLKSIZE.1000)(A))OUTPUT)
```

The second and third argument of this open indicates that the data set has a record length of 100 bytes, a block size of 1000 bytes, that the first character in each record is a control character, and that the data set is going to be used for output. The record length and the blocksize can be given on the DD card instead of in open. All other DCB parameters are fixed by open and they cannot be changed by a LISP user. The record format is set to fixed blocked, and the error option is 'accept' on input, and 'skip' on output.

The three ddnames LISPIN, LISPOUT, and LISPUNCH are given special significance in open. LISPIN and LISPOUT are opened automatically by the interpreter and therefore need not be opened. The second and third argument is implied by LISPUNCH, and they are therefore ignored when open

is given LISPUNCH as its first argument. LISPUNCH implies a record length of 80 bytes, a blocksize of 80 bytes, that the first character in each record is data and not a control character, and that the data set is to be used for output.

One of the atoms SYSIN, SYSOUT, SYSPUNCH and SYSFILE can be used as the second argument of open.

SYSIN implies a record length of 80 bytes, a blocksize of 80 bytes, and that the data set will be used for input.

SYSOUT implies a record length of 133 bytes, a blocksize of 665 bytes, that the first character in each record is a control character, and that the data set will be used for output.

SYSPUNCH implies a record length of 80 bytes, a blocksize of 80 bytes, and that the data set will be used for output.

SYSFILE implies a reocrd length of 80 bytes and a blocksize of 1600 bytes. SYSFILE should be specified for all data sets used by chkpoint or restore.

6.1.2 close[ddname]

All data sets should be 'closed' by the function close after use. close takes as its argument the ddname on the DD card that defines the data set. The two ddnames LISPIN and LISPOUT refer to data sets that remain open throughout a LISP job. LISPIN and LISPOUT cannot be closed by close. They are, however, closed automatically at the end of a LISP job.

6.1.3 asa[x]

A control character is normally prefixed to all output records produced by LISP/360. Executing asa[NIL] stops the prefixing of control characters. This is useful when LISP/360 is used to produce output that will be input to LISP/360 later on. Executing asa[T] will cause LISP/360 to start prefixing control characters again.

6.1.4 otll[n]

$0 < n \leq 120$. otll (out-line-length) specifies how many character posi-

tions LISP/360 can use in each output record. LISP/360 will, after otll[n] has been evaluated, fill in exactly n positions in each output record. Atoms will, whenever necessary, be split across two output records so that precisely n positions are filled in each output record. This is useful when LISP/360 is used to produce output that will be input to LISP/360 later on. In a few cases, otll is called automatically by wrs.

6.1.5 wrs[ddname]

wrs (write-select) is an output directing function and takes as its argument the ddname on the DD card that defines the desired output data set. All output from LISP/360 will go to the data set associated with the ddname after wrs[ddname] has been executed. The two ddnames LISPOUT and LISPUNCH are given special significance in wrs. Executing wrs[LISPOUT] will, in addition to directing the output to LISPOUT, have an effect similar to executing asa[T] and otll[100]. Executing wrs[LISPUNCH] will, in addition to directing the output to LISPUNCH, have an effect similar to executing asa[NIL] and otll[72]. wrs will open LISPUNCH if it was not already opened. A data set produced by print when LISPUNCH was write selected is in SYSIN format.

6.1.6 inll[n]

inll (in-line-length) specifies how many character positions LISP/360 should scan in each input record. This is useful when LISP/360 is required to read data sets that are not in SYSIN format.

6.1.7 rds[ddname]

rds (read-select) is an input selecting function and takes as its argument the ddname on the DD card that defines the desired input data set. All input to LISP/360 will be taken from the data set associated with the ddname after rds[ddname] has been executed. The ddname LISPIN is given special significance in rds. Executing rds[LISPIN] will, in addition to selecting input from LISPIN, have an effect similar to executing inll[72].

6.2 Checkpoint Facilities in LISP/360

Freecellstorage and binary program space can be preserved at any time by executing chkpoint. Freecellstorage and binary program space can then be reset to the state it was in when preserved by executing restore. chkpoint and restore should only use data sets opened by using the DCB descriptor SYSFILE.

6.2.1 chkpoint[ddname]

Execution of chkpoint[ddname] will cause freecellstorage and binary program space to be written into the data set associated with the ddname. A data set created by chkpoint under LISPA requires about 30 tracks on an IBM 2314 disk pack. Only the data sets associated with LISPIN, LISPOUT, LISPUNCH and the ddname given as an argument to chkpoint should be open when chkpoint is executed.

6.2.2 restore[ddname]

Execution of restore[ddname] will cause freecellstorage and binary program space to be overwritten by the contents of the data set associated with the ddname. restore will check whether the data set is compatible with the LISP system that executes restore. A data set checkpointed under LISPA cannot be restored under LISPB or LISPC, or vice versa. The function bpsz must be used with caution when chkpoint or restore appear in the same run as bpsz. A data set created by chkpoint can only be restored when the binary program space has the same size as when the chkpoint was executed. The LISP compiler and LAP will be made available as checkpointed data sets. Only the data sets associated with LISPIN, LISPOUT, LISPUNCH and the ddname given as an argument to restore should be open when restore is executed.

7. The LISP Compiler and Assembler

The addition of the LISP assembler (LAP) and compiler can decrease the running time of a LISP program (formerly run interpretively) by a factor of from eight to twelve depending upon the particular application. However, the theoretical differences between compilers and interpreters impose certain restrictions on what can be compiled. These restrictions are easily bypassed and are mentioned below so that the user will be aware of them as they come up.

The compiler itself calls upon the LISP assembler so that once a function is compiled it is immediately available for execution. LAP was written to closely resemble the OS Assembly Language on the IBM System/360 with certain modifications. It should be remembered that LAP is not only used by the compiler but may be used independently by the LISP user.

7.1 LISP Assembly Program (LAP)

7.1.1 Differences Between LAP and OS Assembly Language

Of the instructions available in OS Assembly Language, a select few have been cmitted -- it was felt that they were unnecessary for LISP users. These were: Set Program Mask (SPM); Set System Mask (SSM); Supervisor Call (SVC); Start I/O (SIO); Test and Set (TS); Test Channel (TCH); Test I/O (TIO); Read Direct (RDD); Write Direct (WRD); Insert Storage Key (ISK); Set Storage Key (SSK). While these instructions are not directly available they still may be generated via the "Define Constant" (DC) instruction described later. Also no extended mnemonics are available. All sixteen of the registers are available in LAP but they must be referenced with an R prefix, i.e., R0, R1 ... R14, R15. In addition one may refer to registers R8, R9, R10 as A, Q, M respectively; R5 as NILR; R4 as K4; R15 as PDL; and R7 as PDS. These aliases will become clear as LAP is described.

Perhaps the major difference between LAP and OS Assembly Language is the availability of quote cells and special cells. Quoted and specialed cells are assembled as pointers to the particular quantities they represent. (See the LISP 1.5 programmers manual) These will be used in examples later so that the user may become familiar with them. Care must be taken in using them. Macros have been prepared to aid in their use.

"Define Constant" and "Address Constant" are defined in LAP in a limited form. They may appear as (DC -logical number-) or (AC -name of location-). No multiplicative factors or variations are allowable. DC's and AC's must be on full word boundaries and this is done in LAP by assembling a NO-OP in front of them if necessary. If the user desires other instructions on full word boundaries he may specify (CNOP) which inserts a half word NO-OP instruction (BCR R0 R0) if necessary to put the next instruction on a full word boundary. Also a reference to an "immediate" field, such as an MVI, can only be a logical number, e.g., (MVI 4(R1), OBX).

There is no indirect referencing in LAP such as the use of * and * + 4,

etc., in L A,* + 4 or even L A,NAM + 4. That is, all locations referenced must be labeled at the point of reference.

LAP is invoked by calling the routine LAP360. It takes two arguments. The first is a list of LAP instructions, the second is a list of dotted pairs representing an initial symbol table or nil (usually nil, for other uses see the LISP 1.5 programmers manual). The first member of the first argument is a list of three elements; first, the name of the routine being defined; second, the type of function either SUBR or FSUBR; and the third, the number of arguments. After this member comes the rest of the instructions, each enclosed in parentheses.

7.1.2 Passing Arguments to and from LAP Routines

For passing arguments between two user defined routines you may use any technique you prefer. However, since it is sometimes necessary to communicate with the interpreter routines, the following scheme is preferred as it is the method employed by the interpreter. As for the actual call to another routine (once the arguments are established) this is done via a macro *LINK which will be described later.

If there is only one argument, it is passed in register A or R8. If there are two arguments, they are passed in A and Q or R8 and R9. If there are more than two arguments (up to a maximum of twenty-two), there is a reserved area in core twenty words long called ARGS in which you can place the third, fourth, etc., arguments. ARGS may not be referred to directly, but its address is permanently located at eight bytes past R12. Therefore, to store the contents of R0 as the third argument: (L M 8(R0 R12)) (ST R0 0(R0 M)) the value of a function is always returned in A.

7.1.3 General Use of Registers

Although all registers have been defined as usable, care must be taken in the use of some of them. Those of special interest are described below:

R3	- used as base register to cover extent of LAP Routine.
R5 or NILR	- contains NIL and should never be altered from that value, but may be used to store NIL in locations or load other registers with NIL.
A, Q	- as mentioned above are used to pass arguments but may be used freely in routines and need not be restored.
M	- completely free for any general use.
R4 or K4	- contains the number 4. May be used locally but must be restored outside the scope of the immediate routine.
R7 or PDS	- this register has meaning only for the compiler and may be used freely in LAP. It must be restored if it is used in conjuction with the compiler.
R6	- points to the next available free cell. It should never be changed.
R11, R12, R13	- used as base registers for the interpreter -- they must be restored.
R0, R1, R2, R14	- completely free for general use.

It should never be assumed that any free register will be preserved in calling another function, even between two LAP defined user routines.

7.1.4 Macros

7.1.4.1 User Defined Macros

Macros may be defined for LAP by doing a DEFLIST of a LAMBDA definition with the property MC. The LAMBDA definition must have one argument which will become a list of the arguments to the macro. The value of the macro should be a list of instructions to be inserted.
For example:

```
DEFLIST(((*SAVE (LAMBDA (x) (LIST (CONS (QUOTE ST) (CONS (CAR X)
(QUOTE (0 (R7))))) (QUOTE (BXH R7 K4 0 (R12)))))))MC)
```

Then the instruction (*SAVE R15) becomes

```
(ST R15 0 (R7))
(BXH R7 K4 0(R12))
```

Macros may be given any name that the user desires, except, of course, it cannot be the same as a valid instruction mnemonic. The system defined macros all begin with "*" for ease of recognition.

7.1.4.2 System Macros

(*SAVE Rx) - saves register x on an internal push down stack. It should be used with care.

(*UNSAVE Ry) - pops up top item on stack and stores it in register y.

(*SAVE Rx) and (*UNSAVE Ry) are used principally in recursive functions.

(*LOAD Rx (QUOTE...)) - used to load quote cells. Quote cells are in core relative to NIL hence this macro expands to
(L Rx (QUOTE...))
(AR Rx NILR)

(*LOAD Rx (Special Z)) - when loading special cells, the macro expands to
(L Rx (Special Z))
(L Rx 0(NILR Rx))

(*STORE Rx (Special Z)) - for storing special cells. The macro expands to
(L M (Special Z))
(ST Rx 0(NILR M))
NOTE: M is changed when using this macro.

(*RETURN NIL) - used to exit a LAP routine. It branches to a particular place in the interpreter. Expands to (BC 15 48 (R0 R12))
NOTE: *RETURN is the only way to end a LAP routine. "Falling through the end" of a routine is incorrect.

(*LINK FN i) - used to call function FN with i arguments.

Two other macros *MOVE and *REMOVE are used principally by the compiler and will be described in that section.

7.1.5 Sample LAP Program

Define SETC such that (SETC X ((A,1) (X,2) (Y,L)) 7) modifies the second argument to ((A,1) (X,7) (Y,L)) i.e., if the second argument is the ALIST, we are changing the binding of variable X.

```
LAP360(( (SETC SUBR 3)                          1.
         (L M 8(RO R12))                        2.
         (L RO 0(RO M))                         3.
         (ST RO TEMP)                           4.
         (ST NILR 0(RO M))                      5.
         (*LINK SASSOC 3)                       6.
         (L RO TEMP)                            7.
         (ST RO 4(RO A))                        8.
         (*RETURN NIL)                          9.
   TEMP (DC OX)                                10.
                                    ) NIL)     11.
```

Explanation:

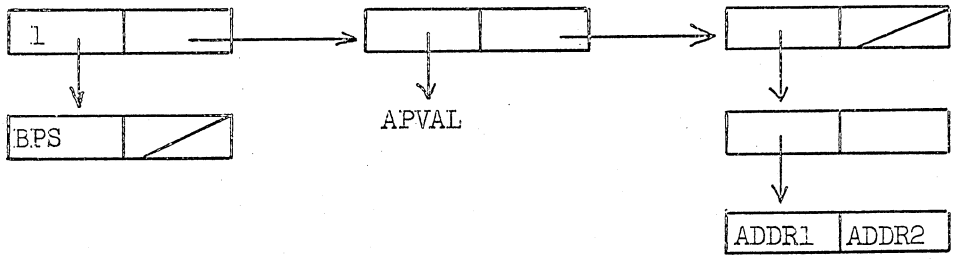
1. Defines SETC as a SUBR with 3 arguments.
2. Picks up the address of ARGS to find the 3rd argument.
3. Puts 3rd argument in RO.
4. Stores RO in temporary location.
5. Sets 3rd argument to NIL.
6. Calls SASSOC which has the same first two arguments as does SETC, hence they remain in A and Q and SASSOC's third argument remains in NIL for this case. SASSOC will return a pointer to the dotted pair whose CAR contains the first argument.
7. Picks up the saved value in RO (this was SETC's 3rd argument), and
8. stores it in CDR of the dotted pair.
9. Returns from the functions. Note that SETC's value is the dotted pair since that is what is in A.
10. Definition of the temporary location.
11. Closes the routine with NIL in the symbol table.

At this point it should be pointed out that the value of LAP360 is the final symbol table of local labels relative to the beginning of the routine in bytes -- hence, in the above example, LAP360 returns ((TEMP.24X)) -- assume that *LINK takes 8 bytes.

7.2 Binary Programming Space

An area is now set aside for binary programs produced by LAP. The size of this area is set when LISP/360 is assembled. However, the area may be eliminated by calling the function bpsz which increases free cell storage. The atom BPS has two pointers indicating how much binary program space is available at any given moment.

The atom BPS mentioned above is slightly different from most atoms as is seen from the following:



ADDR1 and ADDR2 are pointers to the beginning and the end of Binary Program Space, respectively.

7.3 The Compiler

The compiler takes a list of previously defined EXPR's and FEXPR's as its argument and returns a list of the names of the routines compiled. The major restrictions to the compiler are the following:

a. GØ statements within PRØG2's are not allowed.

b. GØ statements within CØND's which are within CØND's are not allowed.

c. Free variables must be declared SPECIAL before compilation. A function called SPECIAL (described later) can be used.

d. Variables used which communicate with the interpreter must be declared CØMMØN before compilation. A function called CØMMØN (described later) can be used.

At the time of publication, these are the major restrictions to the compiler which have been found.

Once compiled, the function is called exactly as it would have been called before compilation.

7.3.1 Auxiliary Routines Available

excise[x]	- one argument. If NIL, the compiler is EXCISED and the space added to free cell storage. If the argument is true the compiler and LAP are EXCISED. One may call excise twice, i.e., EXCISE (NIL) EXCISE (T).
ovoff[ ]	- no arguments. In compiling, a TYPE 8 over or underflow error may occur frequently. This is not an error but ovoff will stop the error message from printing.
ovon[ ]	- no arguments -- restores overflow message.
special[l] & unspecial[l]	- takes a list of variables as arguments and gives or takes away the property "special" to each of them.
common[l] & uncommon[l]	- save as above, except that the property "CØMMØN" is used.

bpsz[ ] - no arguments. Returns all BPS to free cell storage (for jobs requiring a lot of free cell storage and not needing the compiler or LAP).

7.3.2 Examining the Compiled Code

If the user wishes to see the code produced by a compiled function he can do this by saying TRACE((ASSEMBLE)) before the compilation. Two compiler macros *MOVE and *REMOVE will be noticeable in all compiled routines. These set up and restore the push down list upon entering and leaving the routines. The user will also notice many BAL's to a number of bytes past R12. These are interpreter defined routines to handle things like SPECIAL, COMMON and FUNCTIONAL arguments.

7.3.3 Generating the Compiler

The compiler is defined in LISP as is LAP, therefore, to use a compiled version of the compiler and LAP, the compiler is directed to compile itself and this compiled version is used for all future work. This compiled compiler should be checkpointed onto a data set using LISP/360 data management and then restored whenever necessary.

7.3.4 Names of Compiler and Assembler Routines

The following table is a list of the names of the routines used by the compiler and assembler. Care should be taken in using routines with the same names as these, for if they are redefined by the user, the compiler would call the wrong routine.

LAP360
ASSEMBLE
LABLER
LOCAL
LONG
QTCL
LOOK
REGSET
QSET
SPCL
CHCOMP
OVOFF
OVON
REVERSE
CONC
MAP
MAPCON
COMPILE
ATTACH
CALL
CEQ
COM2
COMBOOL
COMCOND
COMMON
COMP
COMPACT
COMPLY
COMPROG
COMVAL
DELETEL
PAAAONE
PHASE2
PA9
PI1
PI2
PI3
PROGITER
SPECIAL
STORE
UNCOMMON
UNSPECIAL
COM1
COMLIS
LAC
LOCATE
PA1
PA2
PA3
PA4
PA5
PA6
PA7
PA8
PA11
PA12
PA14
PAFORM
PAFORM1
PAIRMAP
PALAM
SELECT

APPENDIX

THE LISP INTERPRETER

```
evalquote[fn;args] = [get[fn;FEXPR] ∨ get[fn;FSUBR]→
                        eval[cons[fn;args];NIL]
                      T→ apply[fn;args;NIL]]

apply[fn;args;a] = [
     null[fn]→ NIL;
     atom[fn]→ [get[fn;EXPR]→ apply[expr;[1]args;a];
                get[fn;SUBR]→ { spread[args];
                                $ALIST:=a;
                                TSX subr[1], 4 } ;
                T→ apply[cdr[sassoc[fn;a;λ [[ ];error[A2]]]];args;a];
     eq[car[fn];LABEL]→ apply[caddr[fn];args;cons[cons[cadr[fn];caddr[fn]];a]];
     eq[car[fn];FUNARG]→ apply[cadr[fn];args;caddr[fn]];
     eq[car[fn];LAMBDA]→ eval[caddr[fn];nconc[pair[cadr[fn];args];a]];
     T→ apply[eval[fn;a];args;a]]

eval[form;a] = [
     null[form]→ NIL;
     numberp[form]→ form;
     atom[form]→ [get[form;APVAL]→ car[apval[1]];
                  T→ cdr[sassoc[form;a;λ [[ ];error[A8]]]]];
     eq[car[form];QUOTE]→ cadr[form];[2]
     eq[car[form];FUNCTION]→ list[FUNARG;cadr[form];a];[2]
     eq[car[form];COND]→ evcon[cdr[form];a];
     eq[car[form];PROG]→ prog[cdr[form];a];[2]
     atom[car[form]]→ [get[car[form];EXPR]→ apply[expr;[1]evlis[cdr[form];a];a];
                       get[car[form];FEXPR]→ apply[fexpr;[1]list[cdr[form];a];a];
                       get[car[form];SUBR]→ { spread[evlis[cdr[form];a]];
                                              $ALIST:=a;
                                              TSX subr,[1] 4 } ;
                       get[car[form];FSUBR]→ { AC:=cdr[form];
                                               MQ:=$ALIST:=a
                                               TSX fsubr,[1] 4 } ;
                       T→ eval[cons[cdr[sassoc[car[form];a;λ [[ ];error[A9]]]];
                                    cdr[form]];a]];
     T→ apply[car[form];evlis[cdr[form];a];a]]

evcon[c;a] = [null[c]→ error[A3];
              eval[caar[c];a]→ eval[cadar[a];a];
              T→ evcon[cdr[c];a]]

evlis[m;a] = [null[m]→ NIL;
              T→ cons[eval[car[m];a];evlis[cdr[m];a]]]
```

---

[1] The value of get is set aside. This is the meaning of the apparent free or undefined variable.

[2] In the actual system this is handled by an FSUBR rather than as the separate special case shown here.

REFERENCES

[1] Mc Carthy, J., et al. - LISP 1.5 Programmer's Manual. MIT Press. Cambridge, Massachusetts (1966).

[2] Kent, J. G. - Interpretive System for the Programming of Recursive Functions on a Digital Computer. Intern Rapport E-88. Norwegian Defence Research Establishment, Kjeller, Norway (1966).

[3] IBM, Corp. - IBM System/360 Principles of Operation. Form number: A22-6821-5 (1966).

[4] Kent, J. G. - "LISP 1.5 Implementation on the CD 3600 and the IBM System/360 Series." Paper in The Nature, Uses and Implementation of the Computer Language LISP, (editor E. C. Berkeley). Information International Inc., 200 Sixty Street, Cambridge, Massachusetts. The expected date of publication is October, 1967.

[5] Bolce, J. F. - LISP/360. The University of Waterloo, Waterloo, Ontario (1967).