LISP/360 REFERENCE MANUAL

FOURTH EDITION

MARCH 1972

CAMPUS COMPUTER FACILITY
STANFORD COMPUTATION CENTER
STANFORD UNIVERSITY
STANFORD, CALIFORNIA

DOCUMENT NUMBER SCC024

PREFACE

This manual is intended to provide the LISP 1.5 user with a reference manual for the LISP 1.5 interpreter, assembler, and compiler on the Campus Facility 360/67. It assumes that the reader has a working knowledge of LISP 1.5 as described in the LISP 1.5 Primer by Clark Weissman, and that the reader has a general knowledge of the operating environment of OS 360.

Beginning users of LISP will find the sections The LISP/360 System, Organization of Storage, Functions, LISP Job Set-up, and LISP/360 System Messages most helpful in obtaining a basic understanding of the LISP system. Other sections of the manual are intended for users desiring a more extensive knowledge of LISP.

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

Included in this edition is information on the use of the time-shared LISP system available on the 360/67 which was implemented by Mr. Robert Berns of the Campus Computer Facility staff.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1. THE LISP/360 SYSTEM

LISP 360 operates under the IBM System/360 Operating System (OS). The operation of the LISP executive is best described as follows:

1. Read a function and list of arguments.
2. Start the timer.
3. Pass the function and list of arguments to the function EVALQUOTE for evaluation.
4. Print the execution time and the value of the function.
5. Start again at step 1.

The LISP system initially consists of a large body of predefined functions and provides the facility to add additional function definitions. Statements in the LISP language are evaluated interpretively by the function EVALQUOTE to determine their value, although some functions (such as COMPILE) are evaluated more for their effect than for their value. A compiler and an assembler are also available.

During execution, LISP data structures (including LISP function definitions) are constructed in Free Cell Storage (FCS). The Push-down Stack (PDS) is used to store program parameters dynamically during recursion.

Other system areas are allocated as Binary Program Space (BPS) to contain the machine code for all compiled functions and as I/O Buffers to be used by OS. The general organization of system memory is given in Figure 1.

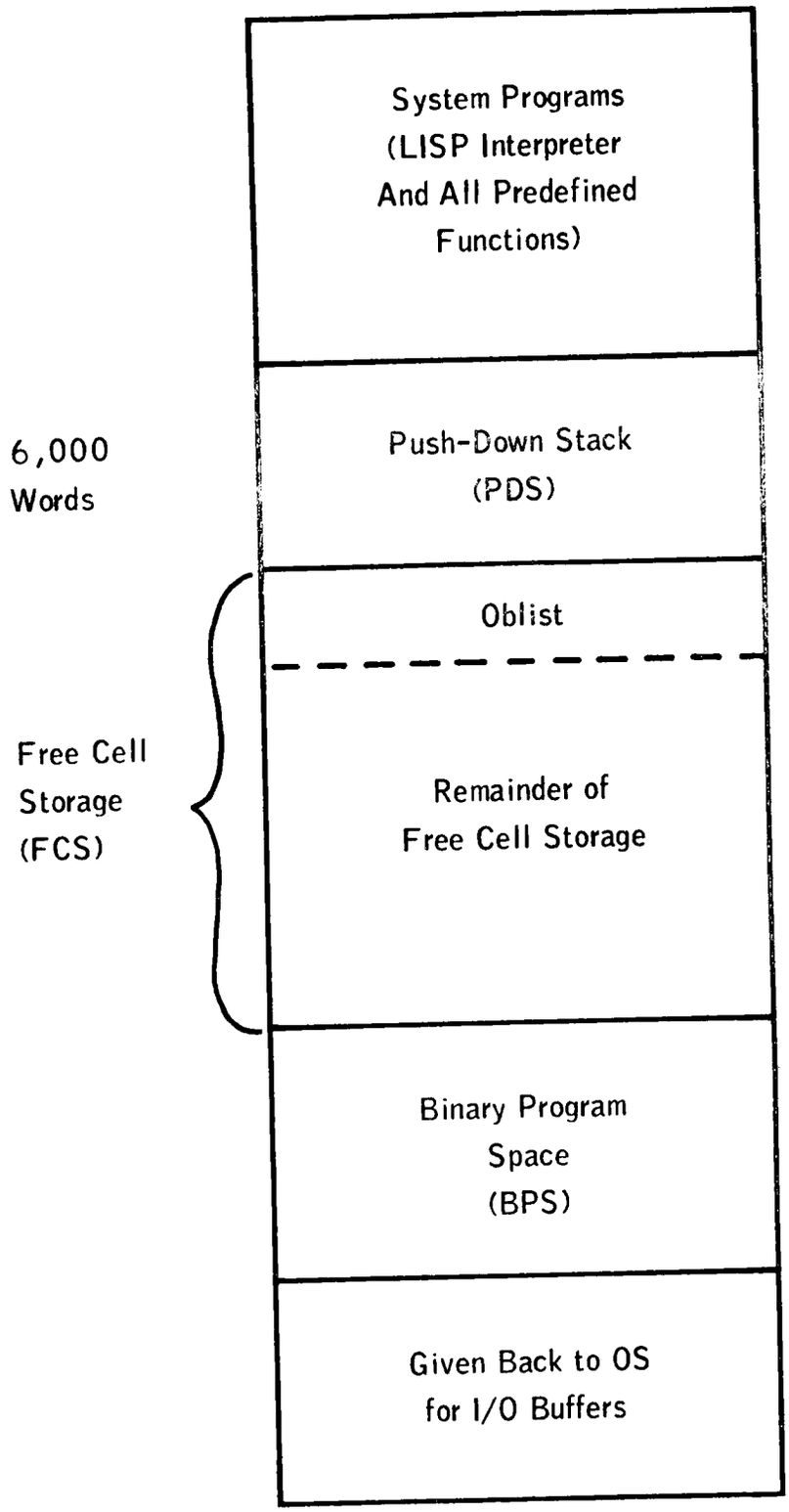

Figure 1: Initial Organization of LISP System Memory

2. ORGANIZATION OF STORAGE

Within the LISP system, computer memory is subdivided into several functional areas. The largest portion of system memory is devoted to Free Cell Storage (FCS), the area used to contain all working data structures. The remaining parts of memory are used for the Push-down Stack (PDS), Binary Program Space (BPS), Input/Output Buffers, and system functions.

2.1 Free Cell Storage (FCS)

A large portion of LISP memory is devoted to the storage of working data structures in Free Cell Storage. Each word of FCS (called a LISP cell) is a System/360 doubleword (64 bits) consisting of an upper word (32 bits) and a lower word (32 bits). LISP cells, depending on their use, may contain four fields as shown in Figure 2.

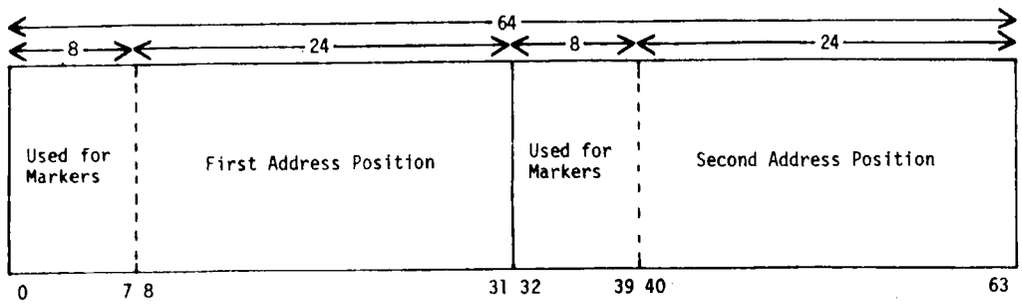

Figure 2: LISP Cell

Initially, all available words in FCS are in a free cell list. As LISP cells are used to create data structures, they are removed from the free cell list until removal of the last word forces the system to perform a garbage collection in an attempt to restore words to the free cell list.

A LISP cell is normally considered to contain pointers to other LISP cells in both its upper and lower words, but a special type of LISP cell is defined in which the upper word contains information other than a pointer. This LISP cell is called a full cell and its format is illustrated in Figure 3.

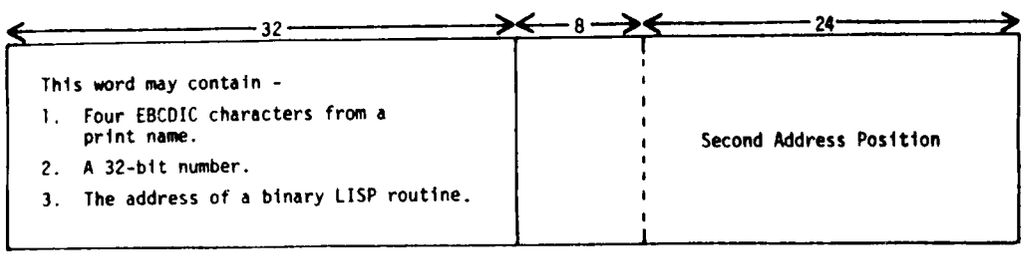

Figure 3: Full Cell

Since the length of the LISP cell is 64 bits and only 24 bits are needed to express an address, the first 8 bits in the upper word and the first 8 bits in the lower word are available for other uses. Figure 4 indicates the uses for some of these bits as explained below.

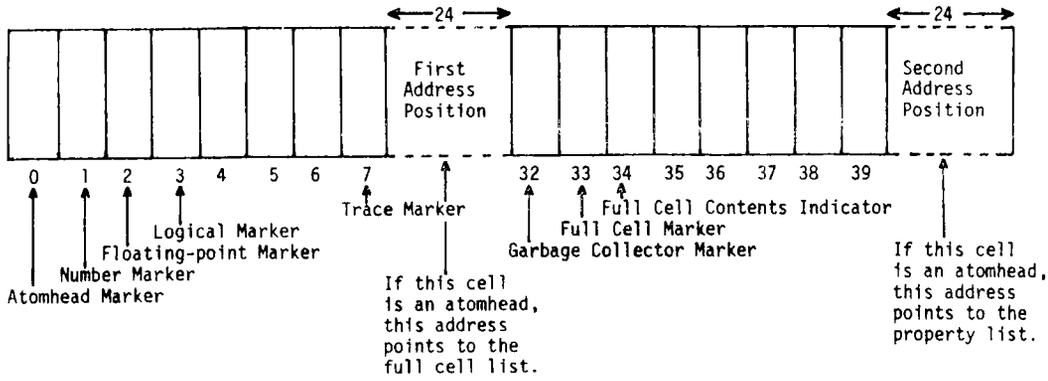

Figure 4: Binary Markers

Bit 0 - indicates that this cell is an atomhead (i.e., the first cell in an atom).

Bits 1, 2 and 3 - refer to a full cell list associated with an atom. Bits 1, 2 and 3 are used as follows:

Bit 1 - Number Marker
Bit 2 - Floating-point Marker
Bit 3 - Logical Marker

For an atomhead (bit 0 is set to one), one of the following bit patterns will be used to describe the full cell list associated with the atom:

bit 0	bit 1	bit 2	bit 3	
1	0	0	0	EBCDIC Characters
1	1	0	0	Fixed-point Number
1	1	1	0	Floating-point Number
1	1	0	1	Logical Number

Bit 7 - indicates that a function is to be traced.

Bit 32 - is used by the garbage collector to mark active cells.

Bit 33 - indicates that this is a full cell.

Bit 34 - is used in a full cell to indicate that the first word (first 32 bits) contains EBCDIC characters or a number. Bit 34 is not set in a full cell when the first word contains an address.

2.1.1 Atoms

An atom begins with a LISP cell (called an atomhead) that contains in its first address position a pointer to a full cell list associated with that atom. The full cell list contains either the printname of the atom (in the case of a literal atom) or the binary value of the atom (in the case of a number).

The second address position contains a pointer to the list of properties associated with that atom -- if it exists (numbers never have properties). The first bit of the first word (bit 0) is set to one to indicate that this cell represents the start of an atom.

Figure 5 illustrates the atom EXAMPLE and its full cell list. The property list is empty.

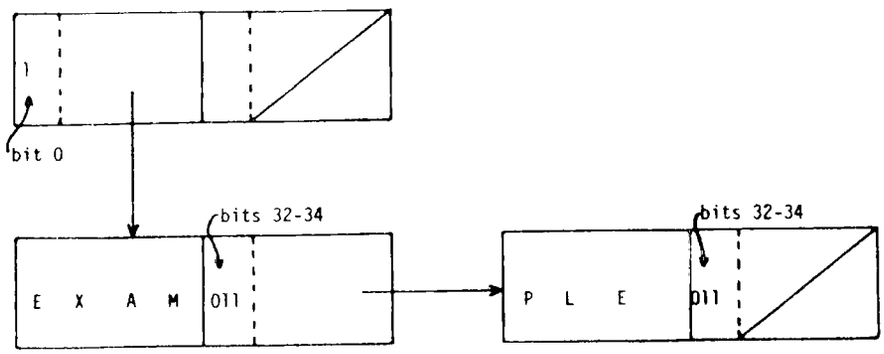

Figure 5: LISP Atom With An Empty Property List

Note: A pointer to the atom NIL is represented by a diagonal line in the address portion of a LISP cell.

Figure 6 illustrates the atom FF and its property list. The property list includes all of the attributes associated with that atom. In this example, the atom FF is a function, namely an EXPR, which starts (LAMBDA . . .)

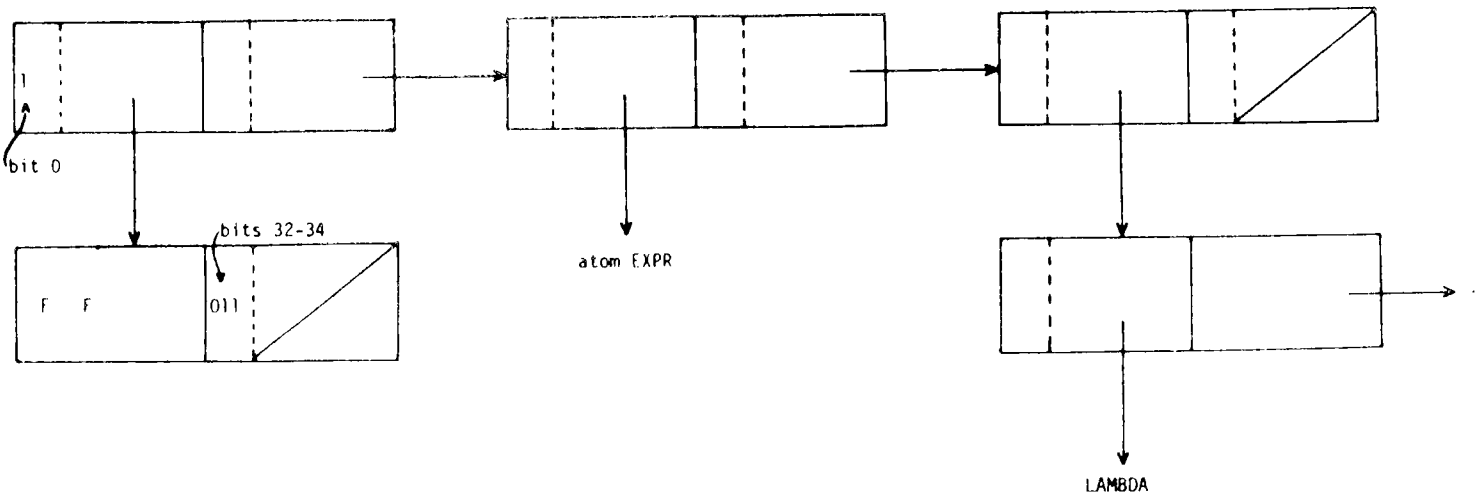

Figure 6: LISP Atom With Associated Property List

Attributes of the atom are designated by flags or indicators on the property list. Flags are atoms which by themselves indicate that the atom (on whose property list the flag occurs) has some attribute (e.g., COMMON). Indicators are atoms which identify the atom (on whose property list the indicator occurs) as having a special value which is found as the next item on the property list (e.g., SPECIAL, APVAL). Indicators used by the LISP system include:

APVAL -- The atom is a constant whose value is the following item in the property list.

EXPR -- The atom is a function name. The lambda expression defining the function is the following item in the property list.

FEXPR -- The atom is a special function name. The lambda expression defining the function is the following item in the property list. An FEXPR differs from EXPR in that the FEXPR is defined with precisely two arguments and may be called with an indefinite number of arguments. When an FEXPR is called, the list of arguments and the current association list are bound to the lambda variables defined in the FEXPR expression, so that the arguments are not evaluated before the function is called.

SUBR -- The atom is a compiled EXPR or a built-in function. The entry address of the subroutine is the following item in the property list.

FSUBR -- The atom is a compiled FEXPR. The entry address of the subroutine is the following item in the property list.

Atoms are created in LISP in several ways. READ, GENSYM1, and MKATOM all create literal atoms. READ creates atoms from the input text and places them on the object list. GENSYM1 creates an atom but does not place it on the object list. MKATOM creates an atom on the object list using the buffer filled by the function RLIT.

Numeric atoms are created by every numeric function. Thus, the same number may be different atoms. These numeric atoms are not placed on the object list.

2.1.2 Numbers

There are three kinds of numbers:

1. Fixed-point (integers)
2. Floating-point
3. Logical (hexadecimal)

All numbers are stored as 32 bit binary numbers with the help of a full cell and must be converted from EBCDIC characters on input and to EBCDIC characters on output. (The EBCDIC representation of a number is not stored.) The first word of a numeric atomhead points to this full cell; the second word is NIL.

A fixed-point number is a signed or unsigned integer (written without a decimal point) in the range -2**31 ≤ number ≤ 2**31-1. For example:

```
0
91
-91
173
-2147483647
```

A floating-point number is a signed or unsigned string of decimal digits with a decimal point. The string of decimal digits may be followed by a decimal exponent. Floating-point numbers may have absolute values in the range 10**-75 ≤ number ≤ 10**75, including zero. For example:

```
7.
-3.4
2.5E+07
-3.2E-4
2.6E7
```

A logical number consists of from 1 to 8 hexadecimal digits (0,1,2,...,9,A,B,C,D,E,F) which may be followed by the letter 'X'. If the number begins with one of the letters A through F, it must be preceded by a zero to avoid ambiguity with character atoms. Logical numbers need not be followed by 'X' if they contain any of the digits A through F. All numeric functions treat logical numbers as integers. For example:

```
14X
-3ABX
0AX
0FFFFFFFCX
14AF5
```

2.1.3 Object List

Pointers to LISP atoms are chained together on a list called the 'object list'. The system searches this list in order to find atoms referenced by the LISP program. The format of the object list is shown in Figure 7. As literal atoms are added to the system, their pointers are added to the front of the object list, immediately following the pointer to the atom NIL except for literal atoms created by GENSYM1, which are not added to the object list. The predefined atom OBLIST has an APVAL on its property list which points to the object list. To print the object list, the following statement can be used: EVAL(OBLIST NIL).

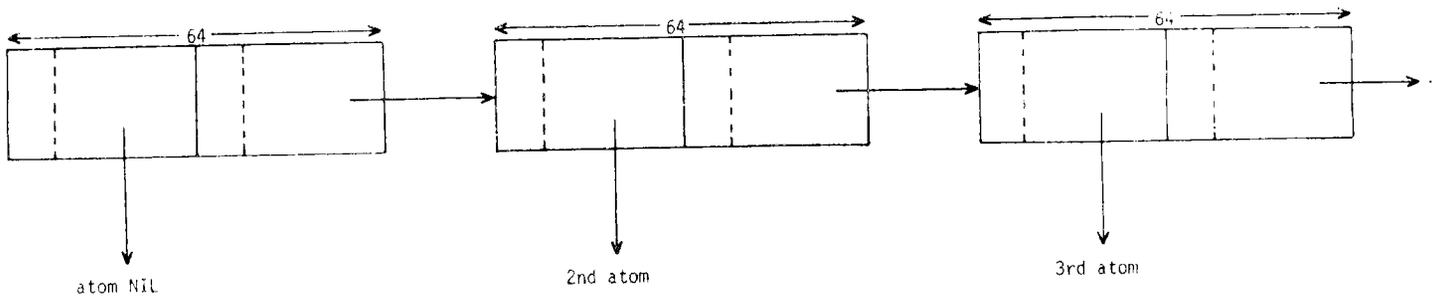

Figure 7: Object List

2.2 Push-down Stack (PDS)

The PDS is used to save active data structures and addresses during program recursion. The size of the PDS is fixed at 6K words (32 bits/word), and it can only be changed by regenerating the LISP system.

2.3 System Functions

The system function area contains the control program, the EVALQUOTE interpreter, predefined system functions, the garbage collector, and the error handler.

2.4 Binary Program Space (BPS)

This area contains all compiled code not part of the standard LISP system (including LAP and the compiler).

2.5 Input/Output Buffers

This is an area of 8K bytes (8 bits/byte) returned to OS for use as input/output storage. The size of the area can be changed any time LISP is loaded by using appropriate EXEC parameters.

3. FUNCTIONS, PREDEFINED ATOMS AND CHARACTER-OBJECTS

3.1 LISP Functions

This section gives the definitions of the functions available in LISP/360. The letters that precede the function names are not part of the function name. They are used to explain the functions as follows:

C - This function is contained in the compiler.

I - This function is contained in the compiler and is for internal compiler usage.

N - This function is not available in time-shared LISP.

T - This function is available in time-shared LISP but not in standard LISP.

The symbols used for function arguments are defined as follows:

alst - association list
at - atom
ch - character-object
ddname - ddname
e - valid LISP form
fn - function
ind - indicator
list - list
n - number
p - predicate
x - S-expression

ADD1(n)

ADD1 takes a number as its argument and returns that number plus 1. If n is a fixed-point number, the result is a fixed-point number. If n is a floating-point number, the result is floating-point.

AND(p1,p2,...,pn)

AND evaluates its arguments from left to right until one is NIL or the end of the list is reached. It returns NIL or T, respectively.

APPEND(list1,list2)

APPEND takes two lists as its arguments. Its value is a list of the elements of list1 followed by the elements of list2.

```
APPEND((A B C) (D E F)) = (A B C D E F)
```

	APPEND1(list,x)	APPEND1 causes the element x to be added onto the end of 'list'; the value is the modified list. APPEND1((A B C) D) = (A B C D)
	APPLY(fn,list,alst)	APPLY causes the function, fn, to be applied to the arguments in the list; alst is used as the association list.
N	ASA(p)	(see Section 6.1.3)
	ATOM(x)	ATOM returns T if x is an atom (either numeric or literal); otherwise it returns NIL.
	ATTRIB(list1,list2)	ATTRIB modifies list1 by tacking on list2 at the end. The value is list2. ATTRIB has the same effect as NCONC although the value is different. Note that if list1 is an atom, list2 is added to the end of the property list of list1.
N	BPSCHKPT(ddname)	(see Section 6.2.3)
N	BPSLEFT( )	(see Section 7.3.2)
N	BPSMOVE(n)	(see Section 7.3.2)
N	BPSRESTR(ddname)	(see Section 6.2.4)
C	BPSUSED(p)	(see Section 7.3.2)
N	BPSWIPE(fn)	(see Section 7.3.2)
N	BPSZ( )	BPSZ takes no arguments. BPSZ deletes all binary program space and adds that storage to Free Cell Storage. Jobs not using the compiler, LAP, or any user compiled functions should call BPSZ for maximum storage. (See Section 7.3.2)
	BREAKP(ch)	BREAKP is a predicate. If its argument is one of these character-objects: blank left parenthesis (right parenthesis) comma , period . its value is T; otherwise its value is NIL.

CAAAR(x)
CAADR(x)
CAAR(x)
CADAR(x)
CADDR(x)
CADR(x)
CAR(x)
CDAAR(x)
CDADR(x)
CDAR(x)
CDDAR(x)
CDDDR(x)
CDDR(x)
CDR(x)

These functions represent all possible nestings of CAR and CDR up to three levels.

N CHKPOINT(ddname) (see Section 6.2.1)

N CLOSE(ddname) (see Section 6.1.2)

C COMMON(list) (see Section 7.3.2)

C COMPILE(list) (see Section 7.3)

C COM1(x1,x2,x3) COM1 is a function used by the compiler.

C CONC(x1,x2,...,xn) CONC is a function used by the compiler.

CONS(x1,x2) CONS obtains a new doubleword from the free storage list (see Section 2.1) and places its two arguments in the first and second words, respectively. It does not check to see if the arguments are valid list structures. The value of CONS is a pointer to the word that was just created. If the free storage list has been exhausted, CONS calls the garbage collector to make a new free storage list and then performs the CONS operation.

N COUNT(n) The argument n must be an integer. COUNT turns on a counter which automatically causes a trap when CONS has been done more than 'n' times. Any CONS performed by system functions are also counted. The counter is turned off by UNCOUNT(NIL). The counter is turned on and reset each time COUNT(n) is executed. The counter can be turned on so as to continue counting from the state it was in when last turned off by executing COUNT(NIL). The function SPEAK( ) gives the current value of the counter, which is decremented each time a CONS occurs.

CSET(at,x)

CSET is used to create a constant by putting the indicator APVAL and a value on the property list of the atom. The value stored in the property list of the atom is CONS(x,NIL). The value of CSET is its first argument. If 'at' already had an APVAL, the old value is removed.

CSETQ(at,x)

CSETQ is like CSET, except that the first argument is quoted instead of being evaluated.

N DEBUG(p)

Currently, this function has no effect.

DEFINE(list)

The argument 'list' of DEFINE is a list of pairs

```
((u1 v1)(u2 v2) ... (un vn))
```

where each u is a name and each v is a lambda-expression for a function. For each pair, DEFINE puts an EXPR on the property list for u pointing to v. DEFINE puts things on at the front of the property list. The value of DEFINE is a list of the u's.

DEFLIST(list,at)

DEFLIST is a more general defining function than DEFINE. Its first argument is a list of pairs as for DEFINE. Its second argument is the indicator that is to be used. The second argument should be a literal atom. After DEFLIST has been executed with (u v) as its first argument, the property list of u will begin with the indicator, at, followed by v.

```
DEFINE(((FN (LAMBDA(X)(CAR X))))) =
 DEFLIST(((FN (LAMBDA(X)(CAR X)))) EXPR)
```

DIFFERENCE(n1,n2)

Both arguments of DIFFERENCE must be numbers. The value is n1 minus n2. If either argument is a floating-point number, the result is floating-point.

DIGP(ch)

DIGP is a predicate. If its argument is one of these character-objects: 0, 1, 2, ..., 9 its value is T; otherwise its value is NIL.

EJECT( )

EJECT takes no arguments. It causes a line to be written with a 'new-page' control character in the first byte (skip to new page).

EQ(x1,x2)

EQ is a predicate which tests if its two arguments point to the same location in storage. Literal atoms are stored uniquely, so that if x1 is an atom, EQ(x1,x2) will be true if x2 is the same atom. List structures and numbers are not stored uniquely, however, and thus it is possible for two equivalent list structures not to be EQ. EQ returns T if its arguments are the same, otherwise it returns NIL.

EQUAL(x1,x2)

EQUAL is a predicate. It returns T if its two arguments are equivalent list structures. EQUAL is recursive, using EQ to test literal atoms. Two numbers are assumed to be EQUAL if they differ by less than 10**-6.

ERROR(x)

ERROR is one way for a user to cause a LISP error. The message '*** a1 - CALL TO ERROR' and the value of x will be printed, followed by a trace-back as described in Section 10.5. ERROR does not return and so it has no value.

EVAL(e,alst)

The first argument e must be a valid LISP expression. It is evaluated using alst as an association list for values of variables.

EVCON(list,alst)

The argument is a list of the form
((p1 e1)(p2 e2)(p3 e3)...(pn en))
where the p's and e's are valid LISP expressions. The p's are evaluated in order until a non-NIL value is obtained. Then the corresponding e is evaluated and its value is returned as the value of EVCON. For each of these evaluations, alst is used as the association list.

EVENP(n)

EVENP returns T if the fixed-point number 'n' is even; otherwise it returns NIL.

EVLIS(list,alst)

The first argument is a list of valid LISP expressions. They are evaluated in order using alst as the association list. The list of the values is returned.

C EXCISE(p)

(see Section 7.3.2)

EXITERR(p)

EXITERR(T) causes the run to terminate after the occurrence of any error that is generated in the execution of the program. EXITERR(NIL), the default, turns off this feature.

EXPLODE(at)

EXPLODE takes an atom as an argument and has as its value a list of the characters in the printname of the atom.

EXPT(n1,n2)

EXPT takes two numbers as its arguments. The second argument must be a fixed-point number. It returns n1 to the n2th power. The value is floating-point if n1 is floating-point or if n2 is negative.

FIX(n)

FIX takes a floating-point number as its argument. The argument is truncated to an integer.

FIXP(x)

FIXP returns T if x is a fixed-point number, otherwise it returns NIL.

FLAG(list,at)

FLAG puts the flag 'at' on the property list of every atomic symbol in the list. Note that 'list' must be a list of atoms. No atom ever receives a duplicate flag. The value of FLAG is NIL.

FLAGP(at1,at2)

FLAGP searches the property list of the atom at1 (CDR at1) for an occurrence of an item EQ to at2. If such an item is found, the value of FLAGP is the rest of the list beginning with that item. Otherwise, the value is NIL.

FLOAT(n)

FLOAT takes a fixed-point number as its argument. It returns that number converted to floating-point.

FLOATP(x)

FLOATP returns T if its argument is a floating-point number; otherwise it returns NIL.

FUNCTION(fn)

FUNCTION is a special form. Its 'argument' must be the name of a function or a lambda-expression. FUNCTION is used to pass functional arguments to other functions. When the form

```
(FUNCTION (LAMBDA(X)...))
```

is evaluated in interpreted LISP, FUNCTION returns the special form

```
(FUNARG (LAMBDA(X)...)) alst)
```

where alst is the current association list. Then the FUNARG form is interpreted by

	APPLY as a function, with the association list taken from alst instead of taking the association list at the time APPLY is called. Thus, FUNCTION, in effect, saves the current association list along with 'fn', so that later calls will use current variable bindings.
C GENSYM( )	GENSYM is a function used by the compiler.
GENSYM1(at)	GENSYM1 creates a new atom whose printname consists of the first four characters of the atom which is passed as its argument, followed by four digits. The atoms that GENSYM1 creates are NOT on the object list, unlike other atoms in the system. Thus, GENSYM1(ALPHA) = ALPH0502 Even if there already exists an atom whose name is ALPH0502, the result of GENSYM1 will be unique.
GET(at1,at2)	GET searches the property list (CDR) of its first argument for an indicator EQ to its second argument. GET then returns the item following the indicator in the property list. If no element of CDR (at1) is EQ to at2, GET returns NIL.
GO(at)	GO is a special form. Its one argument must be a label in the PROG in which GO appears. Its argument is not evaluated. GO causes PROG to branch to the label specified. In compiled LISP, GO cannot appear except as a statement in a PROG, or in the top level of a COND which is a statement in a PROG. Specifically, GO cannot appear within a PROG2 within a COND.
GREATERP(n1,n2)	GREATERP is a predicate which takes two numbers as its arguments. The value is T if the first argument is numerically greater than the second, and NIL if they are equal or the first is less than the second.
N INLL(n)	(see Section 6.1.6)
C LAP360(list,alst)	(see Section 7)

LAST(list)

The argument is a list. LAST returns the tail end of list which contains only the last element:

LAST((A B C D)) = (D)

(This is the list of the last element, not just the last element).

LEFTSHIFT(n1,n2)

LEFTSHIFT takes two numbers as its arguments. The second argument must be a fixed-point number. The word (32 bits) which contains the number given by the first argument is shifted left the number of places specified by the second argument. If the second argument is negative, the first argument is shifted right. The value is a logical number.

LENGTH(list)

LENGTH returns the number of top-level elements contained in the list given as its argument.

LENGTH(((A B C) D (E. F))) = 3

LESSP(n1,n2)

LESSP is a predicate which takes two numbers as its arguments. The value is T if the first argument is numerically less than the second; otherwise it is NIL.

LETP(ch)

LETP is a predicate. If its argument is one of the letters in the range A, B, ..., Z, its value is T; otherwise its value is NIL.

LIST(x1,x2,...,xn)

LIST takes an indefinite number of arguments, and returns a list of those values.

LITP(ch)

= NOT(OR(BREAKP(ch),DIGP(ch))).

LOGAND(n1,n2,...,nk)

LOGAND takes an indefinite number of arguments. LOGAND performs a bit-by-bit logical AND on its arguments and returns the logical number thus produced.

LOGOR(n1,n2,...,nk)

LOGOR is similar to LOGAND, except that it computes the bit-by-bit logical OR of its arguments.

LOGP(x)

It returns T if its argument is a logical number, and NIL otherwise.

LOGXOR(n1,n2,...,nk) — LOGXOR is similar to LOGAND and LOGOR, except that it computes the logical exclusive OR of its arguments.

C MAP(x1,x2) — MAP is a function used by the compiler.

MAPCAR(list,fn) — MAPCAR takes two arguments: the first is a list and the second is a function of one argument. MAPCAR applies the given function first to the CAR of list, then to the CADR of list, and successively to each element of list until the end of the list is reached. MAPCAR returns a list whose kth element is the value of the function applied to the kth element of the list given as an argument.

C MAPCON(x1,x2) — MAPCON is a function used by the compiler.

MAPLIST(list,fn) — MAPLIST takes two arguments: the first is a list and the second is a functional argument. MAPLIST applies the given function first to list, then to CDR list, and successively to each 'tail end' of list, until the end of the list is reached. MAPLIST returns the list of the values of those function evaluations.

MAX(n1,n2,...,nk) — MAX takes an indefinite number of arguments. MAX returns the largest of its arguments. If any of the arguments are floating-point numbers, the result will be floating-point.

MEMBER(x,list) — MEMBER searches the list for an occurrence of an element EQUAL to x. If such an element is found, MEMBER returns T; otherwise it returns NIL.

MIN(n1,n2,...,nk) — MIN takes an indefinite number of arguments, and returns the smallest of them. If any of the arguments are floating-point numbers, the result will be a floating-point number.

MINUS(n) — MINUS takes a number for its argument, and returns the negative of that number.

MINUSP(n) — MINUSP takes a number for its argument; it returns T if that number is less than zero and NIL otherwise.

MKATOM( ) — MKATOM is a function with no arguments. It is used to make atoms out of the information put into the internal

		character buffer by RLIT or RNUMB. MKATOM returns the atom created.
	NCONC(list,x)	The first argument must be a list. NCONC changes the end of 'list' to point to x. In effect, NCONC is like APPEND except that it actually changes its first argument instead of copying it. NCONC returns the modified first argument.
	NOT(x)	NOT returns T if its argument is NIL and NIL otherwise. It is the same as EQ(x,NIL).
	NULL(x)	NULL is the same as NOT(x).
	NUMBERP(x)	NUMBERP is a predicate which returns T if its argument is a number (logical, fixed-point or floating-point); otherwise it returns NIL.
N	OPEN(ddname,list,at)	(see Section 6.1.1)
C	OPTIMIZE(p)	(see Section 7.3.2)
	OR(p1,p2,...,pn)	OR takes an indefinite number of arguments. The arguments are evaluated from left to right until one is reached whose value is not NIL, or the end of the list is reached. OR returns T or NIL respectively.
	ORDERP(at1,at2)	ORDERP imposes an arbitrary canonical order on literal atoms. For character-objects that order is alphabetic; for all other atoms, the order depends on the actual location in storage of the atomhead. ORDERP returns T if the two arguments are EQ or the first comes before the second in this canonical order, and NIL if the first argument comes after the second.
N	OTLL(n)	(see Section 6.1.4)
C	OVOFF( )	(see Section 7.3.2)
C	OVON( )	(see Section 7.3.2)
	PAIR(list1,list2)	PAIR is a function used internally by the LISP system to build association lists. PAIR takes two lists as its arguments. The lists must be of equal length; otherwise an error will occur. PAIR matches the elements of the first argument with the elements of the second argument and returns

a list of dotted pairs; the CARs of the pairs are the elements of the first list and the CDRs of the pairs are the elements of the second list. The list of dotted pairs is in the reverse order of the input lists.

```
PAIR((A B C) (D E F)) =
                        ((C . F)(B . E) (A . D))

PAIR((A B) (C D E))
-----> *** P2 - TOO MANY ARGUMENTS - EXPR
```

C PAIRMAP(x1,x2,x3,x4) — PAIRMAP is a function used by the compiler.

N PLANT(x1,x2)
N PLANTDC(x1,x2)
N PLANTSQ(x1,x2)
N PLANT1(x1,x2)

These functions are used by the compiler to insert code into BPS (Binary Program Space).

PLUS(n1,n2,...,nk) — PLUS takes an indefinite number of arguments. PLUS computes the algebraic sum of its arguments and returns that number. If any of the arguments are floating-point numbers, the result will be floating-point. PLUS( ) = 0.

PRBUFFER(p) — PRBUFFER takes T or NIL as an argument. PRBUFFER(T) will cause READ and READCH to print the input buffer every time a new record 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 necessary to show exactly what was given as input to LISP.

C PRINLAP(p) — (see Section 7.3.2)

PRINT(x) — PRINT takes an arbitrary S-expression for its argument. PRINT causes that S-expression to be written on the output device currently write selected (default LISPOUT).

PRIN1(at) — The argument of PRIN1 must be an atom (numeric or literal). PRIN1 translates its argument into output format and places it in the output buffer. PRIN1 does not terminate the line, however, and sucessive calls to PRIN1 will place the values immediately following each other in the output line.

PROG(list,e1,e2,...,en) PROG is a special form. It provides the capability to perform iteration by allowing looping and the use of temporary variables. The list contains the variables of the PROG required by the statements e1,e2,...,en. PROG variables are initially NIL; they can be reset with the functions SET or SETQ. The "statements" e1,e2,...,en must be either expressions or literal atoms. The literal atoms are used as statement labels. PROG evaluates the statements e1 through en in sequence, unless it comes to the special forms GO or RETURN. When a GO is evaluated, PROG continues evaluation at the statement immediately following the label given in the GO. When a RETURN is evaluated, the expression given in RETURN is returned by PROG. If no RETURN is reached before the last statement, PROG returns NIL.

PROG2(x1,x2) PROG2 takes two arguments and returns the second as its value.

QUOTIENT(n1,n2) Both arguments of QUOTIENT must be numbers. N1 is divided by n2 and the quotient is returned. IF both n1 and n2 are fixed-point numbers, the value is truncated to an integer; otherwise the result is a floating-point number.

N RDS(ddname) (see Section 6.1.7)

READ( ) The execution of READ causes one S-expression to be read from the current input file (as defined by RDS). The value of READ is the S-expression.

READCH(p) 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).

	RECIP(n)	For floating-point numbers, the value is the reciprocal of n. For fixed-point numbers the value is 0.
	RECLAIM( )	RECLAIM causes a garbage collection to occur. The value is NIL.
C	RELINK(x1,x2)	RELINK is a function used by the compiler.
	REMAINDER(n1,n2)	The value of the function is the remainder given when dividing n1 by n2.
	REMFLAG(list,at)	This function removes all occurrences of the flag 'at' (a literal atom used as a flag on atomic property lists) from the property list of each atomic symbol in the list. When the flag is found, the pointer in the preceding element of the property list is modified to delete the flag from the list. The value of REMFLAG is NIL.
	REMOB(at)	This function removes the atom 'at' from the OBLIST. It causes the symbol and all its properties to be lost unless the symbol is referred to by an active list structure. When an atomic symbol has been removed, subsequent reading of its name from input will create a different atomic symbol.
	REMPROP(at,ind)	REMPROP searches the property list of 'at' looking for all occurrences of the atomic symbol 'ind'. If the atomic symbol is found, it is removed from the list along with the succeeding element. Removal is accomplished as described in REMFLAG. The value of REMPROP is NIL.
N	RESTORE(ddname)	(see Section 6.2.2)
	RETURN(x)	This function is used in the PROG feature. RETURN is the normal end of a program. The argument of RETURN is evaluated and this is the value of the program. No further statements are executed.
N	REVERSE(list)	REVERSE causes the top level of list to be reversed. Thus, REVERSE((A (B . C))) = ((B . C) A).
	RLIT(ch)	RLIT 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(ch)

RNUMB 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.

RPLACA(x1,x2)

RPLACA replaces the CAR of the LISP cell x1 with x2. This provides a method of changing list structures without using CONS, and thus creating no new LISP cells. The value is the new x which can be described as CONS(x2 (CDR(x1)).

RPLACD(x1,x2)

RPLACD replaces the CDR of the LISP cell x1 with x2, as described in RPLACA. The value is the new x which can be described by CONS((CAR x1) x2).

SASSOC(x,alst,fn)

SASSOC searches alst, which is a list of dotted pairs, for the pair whose first element is equal to x. If such a pair is found, the value of the function is this pair. Otherwise the value is the function of no arguments, fn.

C SELECT(q,(q1 x1), ... , (qn xn),x)

This function is used internally by the compiler.

SET(x1,x2)

The value of x1 is bound to the value of x2 on the current association list. The value is the value of x2.

C SETC(x1,x2,x3)

This function is used internally by the compiler.

SETQ(x1,x2)

SETQ is like SET except that the first argument is quoted (not evaluated).

N SPEAK( )

SPEAK gives the number of CONS function calls since the CONS counter was last reset.

C SPECIAL(list)	(see Section 7.2.3)
SUBLIS(alst,x)	Alst is a list of dotted pairs, ((u1.v1)(u2.v2)...(un.vn)). The value of SUBLIS is the result of substituting each v1 for the corresponding u1 in x.
C SUBST(x1,x2,x3)	The value of SUBST is the result of substituting x1 for all occurrences of the S-expression x2 in the S-expression x3.
SUB1(n)	The value of SUB1 is n-1.
TERPRI( )	This function terminates the print line.
TIMES(n1, ...,nn)	The value of TIMES is the product of the arguments.
TRACE(list)	The argument of TRACE is a list of functions. After TRACE has been executed, the arguments and values of these functions are printed each time the function is entered. The value of TRACE is NIL.
T TREAD(x)	(see Section 9)
TTAB(n)	TTAB moves the current output cursor to the nth position in the output buffer. Whatever is PRINTed next will appear starting at the given column.
C UNCOMMON(list)	(see Section 7.3.2)
N UNCOUNT( )	UNCOUNT turns off the CONS counter.
C UNSPECIAL(list)	(see Section 7.3.2)
UNTRACE(list)	This function removes TRACEing from all functions in the list. The value of UNTRACE is NIL.
VERBOS(p)	VERBOS controls the printing of garbage collection messages. VERBOS(NIL) turns off the messages and VERBOS(T) turns the messages on. The value of VERBOS is NIL.
N WRS(ddname)	(see Section 6.1.5)

XTAB(n)	XTAB moves the current output cursor 'n' characters to the right. The argument must be a positive integer. Whatever is PRINTed next will appear starting 'n' columns to the right of the end of whatever was last printed (using PRIN1).
ZEROP(n)	ZEROP takes a number for its argument. It returns T if the absolute value of its argument is less than 10**-6, and NIL otherwise.

3.2 Atoms With Initial Values

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

Atom	Value
ALIST	association list
BLANK	blank
BPS	start and end of binary program space (see Section 7.2)
COMMA	,
DASH	-
DOLLAR	$
EQSIGN	=
F	NIL
LPAR	(
NIL	NIL
OBLIST	object list
PERIOD	.
PLUSS	+
RPAR	)
SLASH	/
STAR	*
T	T

3.3 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	O	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. Lower-case letters are translated into upper-case letters. Note that READ does not perform this translation.

4. SPECIAL DIFFERENCES IN LISP/360

In LISP/360 there exist special differences of which the user should be aware.

Several differences pertain to numbers:

1. Fixed-point numbers may have absolute values up to 2**31.

2. Floating-point numbers may have absolute values between 10**75 and 10**(-75), including 0.

3. Floating-point significance on input is 6 digits.

4. Numbers are considered equal if the absolute value of their difference is less than 10**(-6).

5. Signs are ignored in reading logical numbers.

Some other differences refer to atoms, control cards, and several functions:

1. Alphanumeric atoms in LISP/360 may have up to 80 characters.

2. CAR of an atom is not junk as in LISP 1.5, but the address of the full cell list of that atom.

3. No control cards of any type exist in LISP/360.

4. If a PRINT is executed after PRIN1, the list generated by PRINT follows the data output by PRIN1.

5. GO can only be given atomic labels.

6. READ ignores extra right parentheses.

5. LISP JOB SET-UP

LISP statements can be written with a free-field format in columns 1-72. The following control statements are necessary to run the LISP program:

```
//  JOB Statement
/*  KEY Statement  (omit for remote jobs)
//stepname  EXEC  PGM=LISP
//LISPOUT  DD  SYSOUT=A
//LISPIN  DD  *
     .
     .
   LISP Program
     .
     .
/*
```

Additionally, DD statements for using the compiler may be included. An example of these statements is given in Section 7.3.1.

The user may also specify the percentages for allocation of core between free cell storage and binary program space (BPS) in the PARM field of the execute statement. The following statement

```
//stepname  EXEC  PGM=LISP,PARM='F=66'
```

will cause 66 percent of the core available for the run to be allocated to free cell storage and 34 percent of the core to be allocated to BPS. The statement

```
//stepname  EXEC  PGM=LISP,PARM='B=34'
```

will cause the same allocations to be made. If the user specifies both parameters, the 'B' parameter will take precedence. The default values are F=66 (B=34). Thus, if a user is running interpreted LISP only and is not using the compiler, 'B=0' will give the user considerably more core than the default values.

If the user RESTORE's from any file (including the compiler), the values specified in the PARM field are overridden by the values specified when that file was created. In this case, the F and B options of the PARM field are meaningless.

One additional PARM field entry may be made to indicate the amount of core to be reserved by the system for opening and closing files. The statement

```
//stepname  EXEC  PGM=LISP,PARM='R=8K'
```

will cause 8*1024 bytes to be reserved for OS OPEN's and CLOSE's. This parameter may also be specified without the 'K'. For example, R=7000 will reserve 7000 bytes. The default value for 'R' is 8K.

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. The handling of its buffers can be modified by the functions ASA, INLL, and OTLL. It is assumed in the following paragraphs that the reader has a working knowledge of OS/360 Data Management.

6.1.1 OPEN(ddname,list,at)

All data sets must be 'opened' by the function OPEN before they are used. A DD statement is used to define the data set and OPEN uses the ddname in the statement to refer to the data set. The ddname is the argument of OPEN. The record length (LRECL), 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).

The following is an example of the opening of the data set defined by the DD statement named DATA:

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

The second and third arguments of this OPEN indicate 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 to be used for output. The record length and the blocksize can be given in the DD statement instead of in OPEN. All other DCB parameters are fixed by OPEN and they cannot be changed by the LISP user. The record format is set to fixed blocked (FB), and the error option (EROPT) is 'accept' (ACC) on input and 'skip' (SKP) 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 arguments are implied by LISPUNCH, and 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 may 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 record length of 80 bytes and a blocksize of 1600 bytes. SYSFILE should be specified for all data sets used by CHKPOINT or RESTORE.

OPEN puts an APVAL on the atom which is the filename, with a pointer to the DCB for that file.

6.1.2 CLOSE(ddname)

All data sets should be 'closed' by the function CLOSE after use. CLOSE takes as its argument the ddname in the DD statement 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(p)

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)

For 'n' in the range $0 < n < 120$, OTLL (out-line-length) specifies how many character positions LISP/360 can use in each output record. After OTLL(n) has been evaluated, LISP/360 will fill in exactly 'n' positions in each output record. Whenever necessary, atoms will 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 that takes as its argument the ddname from the DD statement 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. In addition to directing the output to LISPOUT, executing WRS(LISPOUT) will have an effect similar to executing ASA(T) and OTLL(100). Similarly, in addition to directing the output to LISPUNCH, executing WRS(LISPUNCH) will have an effect similar to executing ASA(NIL) and OTLL(72). For all other files, the user must call OTLL explicitly - it does not occur automatically. WRS will open LISPUNCH if it is not already opened. A data set produced by PRINT when LISPUNCH was write selected (i.e., WRS(LISPUNCH)) 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 that takes as its argument the ddname from the DD statement 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. In addition to selecting input from LISPIN, executing RDS(LISPIN) will have an effect similar to executing INLL(72). For all other files, the user must call INLL explicitly.

6.2 Checkpoint Facilities in LISP/360

Free cell storage (FCS) and binary program space (BPS) can be saved at any time by executing CHKPOINT. By executing RESTORE, free cell storage and binary program space can then be reset to the state they were in when saved. CHKPOINT and RESTORE should only use data sets that were opened by using the DCB parameter SYSFILE.

6.2.1 CHKPOINT(ddname)

Execution of CHKPOINT(ddname) will cause free cell storage and binary program space to be written into the data set associated with the ddname. 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 free cell storage 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. 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.

6.2.3 BPSCHKPT(ddname)

BPSCHKPT(ddname) is essentially the same as CHKPOINT(ddname) except that only the binary program space is saved.

6.2.4 BPSRESTR(ddname)

BPSRESTR(ddname) reads back into core the data set which was created by BPSCHKPT(ddname).

BPSCHKPT and BPSRESTR make it possible to define multiple BPS areas with different functions in them (using some of the auxiliary functions defined in the next section). Essentially, this means that there is an infinite amount of BPS if the LISP program can be segmented to use compiled functions in logically distinct blocks.

The user is cautioned in using BPSRESTR for two reasons:

1. SUBR pointers in function names are not removed, even if the function is overwritten.

2. BPSRESTR of a function can only be done where free cell storage contains the definition compilation of that function. This is because no maintenance is done on the linkage between free cell storage and binary program space.

7. THE LISP ASSEMBLER AND COMPILER

Use 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 on 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 in the following text so that the user will be aware of them as they arise.

The compiler itself calls the LISP assembler so that once a function is compiled it is immediately available for execution. LAP was written to resemble closely the OS assembler 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 Assembler Language

Of the instructions available in the OS assembler language, the following have been omitted from LAP:

Set Program Mask (SPM)	Set System Mask (SSM)
Test I/O (TIO)	Start I/O (SIO)
Test and Set (TS)	Test Channel (TCH)
Read Direct (RDD)	Write Direct (WRD)
Set Storage Key (SSK)	Insert Storage Key (ISK)
Supervisor Call (SVC)	

While these instructions are not directly available, they still may be generated by use of the 'Define Constant' (DC) instruction. 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, the user may refer to registers R8, R9, and R10 as A, Q, and M, respectively; R5 as NILR; R4 as K4; R15 as PDL; and R7 as PDS. These aliases will become clear as LAP is described.

The major difference between LAP and OS assembler language is the availability of QUOTE cells and SPECIAL cells. These cells are assembled as pointers to the particular quantities they represent. Care must be taken in using QUOTE and SPECIAL cells. Examples are included in this section that illustrate the use of these cells. Also, 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 -S-expression-). No duplication factors or variations are allowed. AC is assembled as the address of the atom minus the

address of NIL. As the garbage collector has no way of knowing about internals of compiled functions, the expression must be an atom on the OBLIST to prevent it from being collected.

DC's and AC'S must be on fullword boundaries and this is done in LAP by assembling a NO-OP in front of the constants, if necessary. If the user desires other instructions on fullword boundaries, he may specify (CNOP) which inserts a halfword NO-OP instruction (BCR R0 R0), if necessary, to put the next instruction on a fullword boundary. Also, a reference to an 'immediate' field, such as an MVI, can only be a logical (hexadecimal) number. For example, (MVI 4(R1), 0BX).

There is no indirect referencing in LAP. The use of * or *+4, etc., (e.g., LA *+4 or LA NAM+4) is not allowed. 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). 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

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

If there is only one argument, it is passed in register A (alias for R8). If there are two arguments, they are passed in A and Q (aliases for 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 the user can place the third, fourth, etc., arguments. ARGS may not be referenced directly, but its address is permanently located at eight bytes past R12. Therefore, to store the contents of R0 as the third argument, code

```
(L M 8(R0 R12))  (ST R0 0(R0 M))
```

The value of a function is always returned in register A.

7.1.3 Register Usage

Although all registers have been defined as usable, care must be taken in the use of some of them. The following describes those of special interest:

R3	- is used as a base register to cover the extent of the LAP routine.
R5 (NILR)	- contains NIL and should never be altered from that value. It may be used to store NIL in locations or to load other registers with NIL.
R15	- is the temporary pointer to the push-down list for compiled code.
R8, R9 (A,Q)	- as mentioned above, are used for passing arguments. These registers may be used freely in routines and need not be restored.
R10 (M)	- is completely free for any general use.
R4 (K4)	- contains the number 4. It may be used locally but must be restored outside the scope of the immediate routine.
R7 (PDS)	- has meaning only for the compiler and may be used freely in LAP. It must be restored if it is used in conjunction with the compiler.
R6	- points to the next available free cell. It should never be changed.
R11, R12, R13	- are used as base registers for the interpreter. They must be restored.
R0, R1, R2, R14	- are completely free for general use.

It should never be assumed that any free register will be saved when 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 the top item on the stack and stores it in register y.

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

(*LOAD Rx (QUOTE...)) - is used to load QUOTE cells. QUOTE cells are in core relative to NIL. Therefore, this macro expands to

```
(L Rx (QUOTE...))
(AR Rx NILR)
```

(*LOAD Rx (SPECIAL Z)) - is used for loading SPECIAL cells. The macro expands to

```
(L Rx (SPECIAL Z))
(L Rx 0 (NILR Rx))
```

(*STORE Rx (SPECIAL Z)) - is used 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) - is used to exit a LAP routine. This macro branches to a particular place in the interpreter. It 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) - is 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(R0 R12))                  2.
         (L R0 0(R0 M))                   3.
         (ST R0 TEMP)                     4.
         (ST NILR 0(R0 M))                5.
         (*LINK SASSOC 3)                 6.
         (L R0 TEMP)                      7.
         (ST R0 4(R0 A))                  8.
         (*RETURN NIL)                    9.
    TEMP (DC 0X)                         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 R0.

4. Stores R0 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 R0 (this was SETC's 3rd argument),

8. and 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.

It should be pointed out here 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

7.2.1 The Atom BPS

An area is 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 indicated in Figure 8.

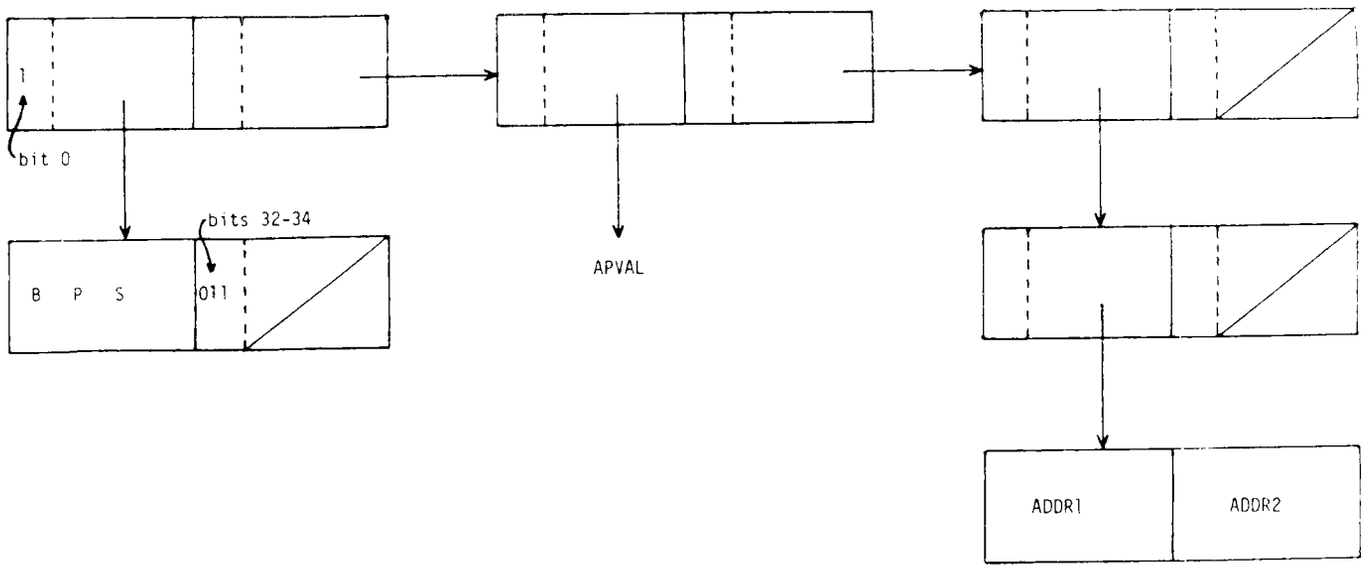

Figure 8: The Atom BPS

ADDR1 and ADDR2 are pointers to the beginning and the end of binary program space, respectively.

7.3 The LISP Compiler

The function COMPILE takes as its argument a list of function names which are EXPR's or FEXPR's. It compiles code in BPS for those functions and replaces the EXPR or FEXPR with an appropriate SUBR or FSUBR property. It returns the list of function names. Functions to be compiled are restricted as follows:

1. GO statements within PROG2's are not allowed.

2. GO statements within COND's which are within COND's are not allowed.

3. Free variables must be declared SPECIAL before compilation. A function called SPECIAL (defined in Section 7.3.2) can be used for this purpose.

4. Variables used which communicate with the interpreter must be declared COMMON before compilation. A function called COMMON (defined in Section 7.3.2) can be used for this purpose.

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

7.3.1 LISP Job Set-up for the Compiler

The following control statements should be used to access the compiler:

```
//    JOB Statement
/*    KEY Statement  (omit for remote jobs)
//stepname  EXEC  PGM=LISP
//LISPOUT  DD  SYSOUT=A
//CMPL  DD  DSN=SYS3.LISPCMPL,DISP=OLD
//LISPIN  DD  *
   OPEN(CMPL SYSFILE INPUT)
   RESTORE(CMPL)
   CLOSE(CMPL)
     .
     .
   LISP Program
     .
     .
/*
```

7.3.2 Auxiliary Routines

BPSLEFT( ) - returns as its value an integer indicating the number of words remaining in BPS.

BPSMOVE(n) - provides the ability to move the current BPS to within 'n' words of the end of all BPS. The space vacated is returned to free cell storage. 'n' must be greater than six and less than the current amount of binary program space left.

BPSUSED(p) - takes one argument. If p is true, LAP and the compiler will print the size of the program compiled. BPSUSED(NIL) turns off this message.

BPSWIPE(fn) - takes as its argument the name of an FSUBR or SUBR previously compiled using the COMPILE function. For example, BPSWIPE(ARGNAME) would cause all functions which have been compiled since ARGNAME and including ARGNAME to be erased from BPS. The next function compiled after BPSWIPE has been called will be located in BPS in the same space in which ARGNAME had been compiled and future compiled functions will follow it.

The use of this function is two-fold. First, it can be used for functions whose use is short-lived enabling them to be erased after some point in the run. Secondly, it can be used in conjunction with the routine BPSCHKPT to create multiple BPS files. Since these BPSCHKPT files may come into use at various times in the run, the SUBR pointers are never destroyed. Therefore, the user must be sure that the function he calls does exist in his current BPS. If not, erroneous results will occur.

BPSZ( ) - takes 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).

COMMON(list)
UNCOMMON(list) - takes a list of variables as arguments and gives or takes away the property 'common' for each of them.

EXCISE(p) - takes one argument. If the argument is NIL, the compiler is EXCISED and the space added to free cell storage. If the argument is true, the compiler and LAP are EXCISED. The user may call EXCISE twice. For example,

```
EXCISE (NIL) EXCISE (T)
```

OPTIMIZE(p) - takes one argument. If the argument is T, the function causes optimization of compiled code. However, it does slow down the compilation process. OPTIMIZE(NIL) is the default.

OVOFF( ) - takes no arguments. In compiling, a type-8 overflow or underflow error may occur frequently. This is not an error, but OVOFF will stop the message from printing.

OVON( ) - takes no arguments. This function restores the overflow message.

PRINLAP(p) - takes one argument. If the argument is true, the LAP produced by the compiler will be printed.

SPECIAL(list)
UNSPECIAL(list) - takes a list of variables as arguments and gives or takes away the property 'special' for each of them.

7.3.3 Examining the Compiled Code

If the user wishes to see the code produced by a compiled function he can do this by saying PRINLAP(T) before the compilation. Two compiler macros *MOVE and *REMOVE will be noticeable in all compiled routines. These macros 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. This area contains interpreter defined routines to handle SPECIAL, COMMON and FUNCTIONAL arguments.

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 will call the wrong routine. Where indicated, the '*_' and '**' are part of the atom's name.

COMMON
COMPILE
COM1
CONC
MAP
MAPCON
OPTIMIZE
OVOFF
OVON
SELECT
SPECIAL
UNCOMMON
UNSPECIAL
**CALL
**COMCOND
**COMPLY
**COMPROG
**PAFORM1
**PA1
**PHASE2
**SPECIAL
**UNSPECIAL

*_ASSEMBLE
*_ATTACH
*_CALL
*_CEQ
*_CHCOMP
*_COM2
*_COMBOOL
*_COMCOND
*_COMLIS
*_COMP
*_COMPACT
*_COMPLY
*_COMPROG
*_COMVAL
*_DELETEL
*_LABLER
*_LAC
*_LAP360
*_LOCAL
*_LOCATE
*_LONG
*_LOOK
*_OPTFN
*_PAFORM
*_PAFORM1
*_PAIRMAP

*_PALAM
*_PASSONE
*_PA1
*_PA2
*_PA3
*_PA4
*_PA5
*_PA6
*_PA7
*_PA8
*_PA9
*_PA11
*_PA12
*_PA14
*_PHASE2
*_PI1
*_PROGITER
*_P12
*_P13
*_QSET
*_QTCL
*_REGSET
*_SPCL
*_STORE

8. THE GARBAGE COLLECTOR

Garbage collection refers to the process by which currently unused LISP cells in FCS are returned to the free cell list. The process is initiated whenever the free cell list is empty.

The first phase invclves marking within the confines of the free cell storage area all LISP cells which are in use as part of some list structure. The group of pointers in the LISP system which reference all active data structures are referred to as base pointers. For each base pointer, the system starts with the LISP cell pointed to by the base pointer and marks all LISP cells reached by chaining through the CAR part or the CDR part (both recursively). All cells having an address within the free cell storage area are marked by turning on bit 0 of the CDR part of the cell. Fullword cells are detected and only their CDR parts are chained through. Cells on common sublists which have already been marked are chained through only once.

The second phase consists of collecting all unused cells and placing them on the free word list. The free cell storage is now traversed linearly. Each cell which is marked has its mark bit turned off. Each cell which is unmarked is placed on the free cell storage list, and the number of cells thus collected is counted.

9. TIME-SHARED LISP AT STANFORD

To use the ORVYL version of LISP, the user must be familiar with the Stanford time-sharing system and with the WYLBUR text-editing facilities.

Once the user has logged on, typing the word LISP in response to a COMMAND? prompt will cause the message 'ENTERING STANFORD/LISP' to be typed. The user is then ready to start a LISP session. The commands which are available are the following:

1. DO <range> — This command causes the <range> indicated to be executed. <range> can be any valid WYLBUR range (e.g., DO ALL, DO 10/LAST, DO 5, etc.). The program to be executed must reside in the WYLBUR working data set.

2. GO — This command causes execution to be continued after an interrupt which was caused by hitting the attention key.

3. SET LONG / SHORT / NONE — When executing a function, LISP will print the following if LONG is in effect:

```
ARGS
    <name of function>
    <list of arguments>
VAL
    <resulting value of function>
```

If the SHORT option is in effect, only the resulting value of the function will be printed.

If NONE is in effect, none of the above will be printed and the only output to the terminal will be from a user call to the PRINT function.

SET LONG is the default option.

4. EVQ — This command provides an immediate mode of execution. For example, if the user types

```
EVQ CAR((A B)) CDR((B C))
```

these two functions will be evaluated immediately as opposed to being executed by a DO command and existing in the WYLBUR data set.

5. EXIT — This command terminates the LISP session.

To facilitate I/O to the terminal, a function called TREAD is available to permit dynamic reading of data from the terminal. TREAD is defined as follows:

TREAD(NIL)	will prompt an '!' and read one S-expression from the terminal. This S-expression will become the value of the TREAD function.
TREAD(T)	assumes that the user has previously executed the function PRIN1. The argument of PRIN1 will then become the prompt in place of the '!'.
TREAD(0)	is a dummy call to TREAD which initializes the input buffer so that the next TREAD will read from a newly prompted line.

As is implied above, more than one S-expression may be typed on an input prompted line and successive use of TREAD will read these expressions consecutively (unless there is an intermittent TREAD(0)).

The time-shared version of LISP has no file I/O capabilities or checkpoint and restore facilities. Therefore, all functions pertaining to these features do not exist in the time-shared version. This also applies for the compiler, as well as for certain other functions which would have no meaning in the time-shared environment.

One additional feature is the use of the character-object '>' to indicate 'put enough right parentheses to balance the left parentheses up to this point'. For example,

```
CAR(((((X Y)))))
```

may be written as

```
CAR(((((X Y>
```

To use '>' for other purposes, use $$$>$.

9.1 Example of a Terminal Session

The following is an example of a simple LISP program using the time-shared LISP system available on the 360/67 at Stanford. The program finds the last element of a list. Text typed in all upper case letters indicates system responses and prompts. Lower case letters have been used to indicate information typed by the user.

```
STANFORD  33  10/18/71 12:06:34
NAME? 'w. woodpecker'
ACCOUNT? ■■■■
KEYWORD? ■■■
TERMINAL? p00
COMMAND? set terse

? lisp
:ENTERING STANFORD/LISP

? collect 10 by 10
  10.   ? define(( (last (lambda(l)
  20.   ?   (cond  (  (null l)  nil)
  30.   ?          (  (null cdr l))  (car l) )
  40.   ?          (  t   (last (cdr l) ) )   )
  50.   ? ) )))
  60.   ? ATTN***

? do all

ARGS
     DEFINE
     (((LAST (LAMBDA (L) (COND ((NULL L) NIL) ((NULL CDR L)) (CAR L))
     (T (LAST (CDR L)))))))
VAL
     (LAST)

? evq last(( a b c d e g h j k l m n p q r s))

ARGS
     LAST
     ((A B C D E G H J K L M N P Q R S))
*** A8-UNDEFINED VARIABLE
 *   CDR
 *   ((L A B C D E G H J K L M N P Q R S))
*** TRACE-BACK FOLLOWS
 *   NIL
 *   ((L A B C D E G H J K L M N P Q R S))
 *   (CDR L)
 *   NULL
 *   ((L A B C D E G H J K L M N P Q R S))
 *   ((L A B C D E G H J K L M N P Q R S))
 *   (((NULL CDR L)) (CAR L))
 *   (((NULL L) NIL) ((NULL CDR L)) (CAR L))
```

```
? mod 30
  30.                (   (NULL CDR L))   (CAR L) )
 ALTERS ?                      i(
  30.                (   (NULL (CDR L))   (CAR L) )
 ALTERS ? cr

? do all

ARGS
      DEFINE
      (((LAST (LAMBDA (L) (COND ((NULL L) NIL) ((NULL (CDR L)) (CAR L))
      (T (LAST (CDR L)))))))))
VAL
      (LAST)

? evq last((a b c d e g h j k l m n p q r s))

ARGS
      LAST
      ((A B C D E G H J K L M N P Q R S))
VAL
      S

? set short
? evq last((z y x w v u t s r q d a))
      A

? logoff
EDITING TIME = 0.06 SECONDS
COMPUTE TIME = 2.06 SECONDS
MEMORY USAGE = 92.14 PAGE-SECONDS
I/O ACTIVITY = 0 UNITS
ELAPSED TIME = 00:04:36
END OF SESSION
```

10. LISP/360 SYSTEM MESSAGES

10.1 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 are printed upon return from EVALQUOTE. The time indicated in the above message gives the time spent in EVALQUOTE not including time spent in garbage collection. The time is in milliseconds.

10.2 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. However, because of the nature of the linkage between compiled functions, once a call by a compiled function to a compiled function has been executed untraced, it can never be traced again.

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 unless that function is a FEXPR or a FSUBR. 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.

10.3 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 free cell storage freed, and the size of the push-down stack at each garbage collection. The printing of this message can be controlled by the function VERBOS.

10.4 Interruption Message

An interrupt supervisor takes care of all program interruptions in LISP/360. See the IBM manual System/360 Principles of Operation for information about System/360 interruptions. The program status word (PSW), the contents of registers 0-15 and the message '***ERROR: CAR TAKEN OF FULLCELL' are printed if the interruption code is 1 to 7. A trace-back is then given of the same type as described in Section 10.5.2. This type of interruption 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. Note that many functions (EQUAL, etc.) which chain through the CDR of lists do not check for the full cell mark. Thus, if these functions are applied to the CAR of an atom or a property list which contains an FSUBR or SUBR, this type of interruption can occur. Additionally, this type of interruption can occur during the trace-back of another error.

An interruption code of 8 to F means that an overflow or underflow occurred. This type of interruption causes the message '***OVER- OR UNDERFLOW OF TYPE xx' to be printed. xx is the interruption code. Execution of the function that caused the overflow or underflow is resumed after the interruption.

10.5 Error Diagnostics

10.5.1 Syntax Errors

If the scanner finds syntactical errors in an S-expression, it inserts special atoms at appropriate places in the S-expression. These special atoms are used as follows:

Atom	Meaning
ERRB	A '.' (dot) encountered as the 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 the 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.

10.5.2 Execution Errors

When an error occurs during execution, the following type of error diagnostic is printed:

```
***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 includes 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. Therefore, the first few lists will usually give a good indication of what caused the error.

As an example, 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 and association lists. When reading the stack, the user should keep in mind that the innermost functions are evaluated first, even though the functions are interpreted from the outside in. Therefore, the call on the function being evaluated when the error occurred will be near the top of the stack, if the call to that function is being interpreted.

If TRACE is executed within a LISP job, the name of an EXPR that was called will be found on the stack between the definition of the EXPR and the corresponding association list. If a function using the PROG-feature was called, it will cause the following lists to appear in the stack printout:

The association list.

The GO-list.

A list of the uninterpreted statements in the function starting with the one to be evaluated when the error occurred.

The complete argument of PROG (omitting the name of the function).

The following is an example of the error that might occur when using the PROG definition shown. After the function has been defined and called, the error messages given below would be printed. Note that the four items after the trace-back message are the ones described above.

```
DEFINE(((TEST2(LAMBDA(X)(PROG(Y)
(SETQ Y (CAR X))
(SETQ Y (CONS X Y) )
(SETQ Y (CAR Y))
(SETQ Z (CAR Y)) ))))))

  TEST2((A B C))
```

After execution has started, the following will appear:

```
***A5-SET VARIABLE UNDEF      (see Section 10.5.3)
 *   Z
 *   ((Y A B C) (X A B C))
***TRACE-BACK FOLLOWS
 *   ((Y A B C) (X A B C))
 *   NIL
 *   ((SETQ Z (CAR Y)))
 *   ((Y) (SETQ Y (CAR X)) (SETQ Y (CONS X Y)) (SETQ Y (CAR Y)) (SETQ
     Z (CAR Y)))
```

10.5.3 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 prepositions 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 were 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

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 an 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-expressions 1 and 2 will, if given, depend on where in the interpreter the stack was last used. The trace-back 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.

GC2-STORAGE EXHAUSTED
The garbage collector is unable to find any unused cells in free cell storage. S-expressions 1 and 2 are the arguments of CONS. The trace-back is not given on this error. This is a fatal error and LISP/360 gives up control to OS.

I3-BAD ARITHMETIC ARGUMENT
An arithmetic routine was given a non-arithmetic argument. S-expressions 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 trace-back 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 (i.e., right parentheses or terminating character in the '$$' notation) generated. The trace-back is not given on this error. This is a fatal error and LISP/360 gives up control to OS.

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 trace-back is not given on this error. The error causes LISP to start reading from LISPIN.

R5-NAME OR NUMBER TOO LONG
An EBCDIC printname or a number is longer than that accepted by the interpreter. Truncation occurs on the right. Only the message appears for this error.

APPENDIX

THE LISP INTERPRETER

```
evalquote[fn;args] = [get[fn;FEXPR] V 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;¹args;a];
                              { spread[args];³
              get[fn;SUBR] -> { ALIST:=a;      ;
                              { BAL subr¹
              T -> apply[cdr[sassoc[fn;a; [[ ];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[apval¹];
          T -> cdr[sassoc[form;a; [[ ];error[A8]]]]];
 eq[car[form];QOUTE] -> 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;¹evlis[cdr[form;a];a];
                    get[car[form];FEXPR] -> apply[fexpr;¹list[cdr[form];a];a];
                                            ⎧spread[evlis[cdr[form];a]];³
                    get[car[form];SUBR] -> ⎨ALIST:=a;                    ;
                                            ⎩BAL subr¹
                                            ⎧A:=cdr[form];
                    get[car[form];FSUBR] -> ⎨Q:=ALIST:=a    ;
                                            ⎩BAL fsubr¹
                    T -> eval[cons[cdr[sassoc[car[form];a; [[ ];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]]]
```

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

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

³'spread' loads the appropriate registers with the values given it.

Note: Some modification of the definition is necessary where actual machine instructions are shown to give the representation for the IBM System/360.

REFERENCES

1. LISP 1.5 PRIMER, Clark Weissman, Dickenson Publishing Company

2. The Programming Language LISP: Its Operation and Applications, Berkeley, E. C. and Bobrow, D. G., editors, M.I.T. Press

3. LISP 1.5 Programmer's Manual, McCarthy, J., M.I.T. Press

4. Programming Systems and Languages, Rosen, S., editor, McGraw Hill Publishing Company, pp. 455-490

5. An Introduction to LISP, Griffith, A. K., University of Florida

6. The BBN-LISP System, Bobrow, D. G., Murphy, D. L., and Teitelman, W., Bolt Beranek and Newman, Inc.

7. Stanford LISP 1.6 Manual, Quam, L. H., Stanford Artificial Intelligence Project

8. A Paged LISP Using the Dynamic Relocation Hardware of an IBM 360/67, Berns, R. I., (soon to be published)

9. IBM System/360 Principles of Operation, Form No. A22-6821