UNIVERSITY of PENNSYLVANIA
PHILADELPHIA 19104

The Moore School of Electrical Engineering **D2**
DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

May 22, 1978

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139

Attn: Dr. Guy L. Steele, Jr.

Re: UP-MS-CIS-76-34 - LISP 7/16 Implementation Report

Dear Dr. Steele:

We are pleased to enclose herewith the technical report that you requested on May 14, 1978.

Very truly yours,

G. Yerkes

G. Yerkes

gy
enc.

UP-MS-CIS-76-34

LISP 7/16 Implementation Report

Thomas Kaczmarek

LISP 7/16 - LISP on the Interdata

This document is designed to describe the implementation of LISP on the Interdata 7/16 minicomputer. It is organized into two major sections. The first of these will describe the representations used for the various types of data. The second will describe the various modules which are used in the LISP interpreter. Throughout this document it will be assumed that the reader is familiar with LISP. This document is not a user's guide; rather it is intended as a guide for maintaining and or modifying the interpreter.

DATA STRUCTURES

Cells

Many of the structures which will be described later will consist of a collection of cells which are linked together. A cell in LISP 7/16 consists of two Interdata halfwords which have been fullword aligned. The fullword alignment is done to force pointers to have values which are multiples of four. This frees the two low order bits for marking purposes. How these bits are used will be explained later. All of the cells in the system are contained in an area of the program that is identified by the label SPACE. The label BOTSPC refers to the first cell in SPACE (the cell with the lowest address). The label TOPSPC refers to the last cell in SPACE. The variable TOPSP holds a pointer to TOPSPC. During execution, whenever any module needs to know where the last cell is, it checks TOPSP. The size of SPACE may be modified by changing the value stored in TOPSPC. When LISP is executed, the program begins by linking together all of the cells in SPACE which are not part of predefined structures. The predefined structures are used to hold the OBLIST, the definitions of predefined atoms and the hash lists for the print names of the atoms. (These will all be described later.) The label STRTSP is used to refer to the first available cell at initialization time. Every time the LISP program is started at its starting address, the list of available cells is initialized. As mentioned above, each cell consists of two halfwords. The first of these, the one with the lower address, is referred to as the CAR of the cell; and the second is referred to as the CDR of the cell.

Atoms

Literal atoms -

A literal atom in LISP 7/16 is represented as a single cell. It is marked as an atom by setting the bit corresponding to a x'2' in the CAR of the cell. If the CAR of any cell has this bit set, it is considered to be a literal atom. The CAR of the atom is x'FFFE' unless it has a value assigned to it by either the SET function or through a LAMBDA binding. If a literal atom has a value assigned then the CAR of the cell contains a pointer to the value plus two. The plus two is a result of the x'2' bit being set. Past values of an atom are stored on a stack whenever a LAMBDA binding has taken place. LAMBDA bindings will be described more fully later. The CDR of an atom points to the property list of the atom. If there is no property list, the CDR points to the special atom NIL. The property list and NIL will be described later.

Fixed-point numeric atoms -

Fixed-point numeric atoms consist of one cell and may contain values between -32768 and 32767, i.e. 16-bit two's complement integers. The CAR of a fixed-point atom always points to itself. Any cell whose CAR is a pointer to itself is considered to be a fixed-point atom. The CDR of the atom contains the value in 16-bit two's complement notation.

Floating-point numeric atoms -

Floating-point numeric atoms consist of three cells. The floating-point atom has the structure of a dotted pair of fixed-point atoms. Any structure that looks like a dotted pair of fixed-point atoms will be treated as a floating-point atom. The CAR of the floating-point atom points to a cell whose CDR contains the 16 high order bits of the value. The CDR of the atom points to a cell whose CDR is the 16 low order bits of the value. The value of the floating-point atom may be obtained by concatenating the low and high order bits. After performing this operation, the result is in standard Interdata single-precision floating-point format.

The OBLIST

The OBLIST in LISP 7/16 is a list which begins at the label POBLST. The OBLIST points to each atom in the system through its print name (pname). The OBLIST is a list of 16 lists. The 16 sublists are hash buckets which are accessed by using the four low order bits of the ASCII representation of the first character in the print name of the atoms. Thus, the first list points to all atoms which begin with the letter 'P', since the first character must be a letter; and 'P' is the only letter with an ASCII representation ending in four zeroes. The second list points to all atoms which begin with the letter 'A' or 'Q', having ASCII representations of x'41' and x'51' respectively. Each of the 16 sublists consists of pointers to the print names of the atoms which are stored in the pname area. The section following will describe the print names and the pname area. Figure 2 in Appendix C illustrates the relation between atoms and the OBLIST.

Print Names

The print names in LISP 7/16 are not cells. They are stored in a special area referred to by the label PNAMES. The label LSTOB refers to the first available space in this area upon initialization. The variable TOPOB is initialized to point to this location and is updated as the space is filled up. LISP

7/16 checks to be sure that LSTOB never exceeds BOTSPC which is the first location above PNAMES. The individual pnames are stored as variable-length items in this area. The first byte of the pname tells the number of characters in the pname. It is followed by the ASCII characters in the print name. The characters are followed by a pointer to the atom. This pointer is halfword aligned on the next available halfword. One byte of zeroes is placed between the ASCII characters and the pointer if necessary. Refer again to Figure 2 in Appendix C for an illustration.

Property Lists

The property lists in LISP 7/16 are pointed to by the CDR of the literal atom they are associated with. Property lists are always of even length. If the elements of the list are numbered 1,2,3,...,2n, then the odd elements of the list (1,3,...,2n-1) are taken as pointers to the property indicators. The even numbered elements (2,4,...,2n) point to the values associated with those indicators preceding them in the list.

The Stack

The stack in LISP 7/16 is used to save function calls and argument bindings. The stack begins at the location PSHDN, and TOPCOR is the last location available to the stack. The variables PUSHD and TOPCR are used to hold pointers to these locations. Every time a SUBR is executed, an element is pushed onto the stack. This element consists of eight halfwords. These halfwords are the contents of registers 8 through 15 upon entry to the SUBR. Registers 8 through 10 point to the arguments of the SUBR. There may be at most three arguments passed. Register 11 is used to point to structures which are to be saved in case of garbage collection. This is used by SUBR's which build up structures and must protect the structures being built. Register 12 is used to link the stack elements, and it tells where the next element may be placed. Register 13 points to the atom which is defined as the SUBR being executed. Register 14 contains the return address and register 15 is used as a pointer to the last element on the stack. Thus the stack is doubly linked. The pointers to the last and the next element on the stack are needed since the stack is also used to hold LAMBDA bindings and these are not of fixed length. LAMBDA bindings are stored as a pair of halfwords on the stack. One pair is used for each variable which is bound. The first of these is a pointer to the atom which is having a value bound to it, and the second points to the old value of the atom. (Recall that the CAR of the atom points to the current value.) Figure 3 in Appendix C depicts the stack.

NIL

NIL is a special literal atom. It is represented by the value zero. Any pointer with a value of zero is considered to be a pointer to NIL.

LISP 7/16 MODULES

This section of this document will describe each of the modules which make up the executable code of LISP 7/16. The modules will be described in the order in which they appear in the program. The program has three types of modules: interpreter modules which serve to initialize the program and direct the operation of the basic interpretive cycle; LISP SUBR's which are LISP functions available to the user as machine language routines; and internal subroutines which are used by the various SUBR's either to structure the code or to provide a function which several different SUBR's may have need for. Table 1 in Appendix D lists and classifies these modules.

A set of naming conventions has been adopted in LISP 7/16. LISP atoms are given their print names each truncated to six characters as a label. Thus the cell for the atom PRINT has the label PRINT. The routine to perform a SUBR is given as label the print name of the atom truncated to five characters and suffixed with the character 'S'. Thus the routine to do printing is labeled PRINTS. (RPLACA and RPLACD are exceptions to this rule. The rule would not yield unique names. RPLCAS and RPLCDS are used.) The print name data is labeled by the print name truncated to five characters suffixed with the letter 'P'. Thus the print name of PRINT has the label name PRINTP.

INIT -

Initialization is done by the code beginning at the label INIT and running to the label RESTRT. This code links the cells beginning at STRTSP and ending at the location pointed to by TOPSP, setting the CAR of each cell to zero and the CDR of each cell as a pointer to the next cell. The loop between LOOPIN and ENDLP initializes this list. At ENDLP the CDR of the last cell is set to NIL to mark the end of the list. The object hash list is then initialized. This is accomplished by placing NIL in 16 locations to mark the end of the 16 sublists of the OBLIST mentioned above. INIT then initializes the counter used to generate atoms via the GENSYM function. Finally, INIT initializes TOPOB to point to the next available location in the pname area and saves the pointer to the next available cell in SPACE in case an error causes the pointer maintained in the register NXTAVL to be destroyed.

RESTRT -

RESTRT marks the bottom of the stack by pushing a dummy element with a next element pointer of x'FFFE'. It then resets the pointer to the next available location in SPACE which was saved in INIT. RESTRT is the location that all errors return to, and it may be used to restart the program from the operating system to preserve previously defined functions.

EVALOP -

This loop is the basic interpretive cycle. It does a READ followed by an EVAL followed by a PRINT. The linkage to each of these routines is via the standard linkage conventions used throughout LISP 7/16. These conventions are described in Appendix B.

ADDS - SUBS - MULTS - DIVIDS -

These four labels are dummy entry points into the routine ARITHS. Each of these loads into register CELLPT an index which will be used inside ARITHS to branch to the proper arithmetic instruction.

LESSS - GREATS - EQUALS -

These are also dummy entries to ARITHS. Once again these load the index of the proper instruction in register CELLPT. In this case, however, they all cause the execution of a compare instruction. A branch and link is used when transferring to ARITHS. The arithmetic operators mentioned above do a simple branch into ARITHS. Upon completion of the compare instruction, ARITHS returns to these comparison routines and the condition code is tested. Exit from these routines passes either through RETF or RETT depending on whether the comparison was false or true respectively. If the result is false the atom NIL is returned. If the result is true the atom T is returned.

ARITHS -

Arithmetic operations and comparisons are done by the routine ARITHS. This routine, like all SUBR's that will be described, begins by saving a stack element, saving the number of arguments passed to it and then checking if the number passed is correct. The number of arguments passed is saved in case of an error, and a later section will describe why this is done and how it is used. ARITHS then checks if both arguments passed are fixed-point arguments. If so, it gets the value of each in the appropriate pair of registers and then uses the index passed in CELLPT to branch to the proper instruction. Before doing so, it checks to see if the second argument is negative. If it is, it expands it to a 32-bit fixed-point number by loading a -1 in the appropriate register. This is done because the Interdata multiplication and division instructions require it. If the operation was a compare instruction then ARITHS returns via the link register for testing the condition code. If it is not a compare, a new numeric atom is created using the CONSEM routine. This new atom is returned as the result.

If the first argument passed to ARITHS is a fixed-point atom and the second is not, an error results. If the first argument is not a fixed-point atom, ARITHS tests if it is a floating-point atom. If not, an error is reported. If the first argument is not numeric an error is also reported

If the first argument is not a fixed-point atom, ARITHS branches to TSTFPT. This code tests if the two arguments are floating-point numbers using TSTFLT. This routine requires the CAR and the CDR of the atom in question to be pointed to by registers WORK2 and WORK3, respectively. If they are valid, the operation is performed as mentioned above for the fixed-point case. After the operation is performed and if the operation was not a comparison, a new floating-point atom is created. This is done by using the CRATFP routine. This routine finds the value to use in the location FLTNUM. Thus ARITHS stores the result there before branching and linking to CRATFP. If a comparison operation was performed, ARITHS returns via the link register to have the condition code tested.

CRATFP -

This is an internal routine used to create a new floating-point atom. It gets the value to use from the location FLTNUM, and linkage is via register RTN2. It creates the result by first creating the cell for the low order bits of the floating-point number. It uses the internal routine CONSEM to get the new cell. It saves this partial result in case of garbage collection by making it the CAR of the next cell it will request. This second cell is used to hold the high order bits of the floating-point number. CONSEM is called a second time to obtain the second cell. CRATFP then completes the creation of the floating-point atom by calling the CONSEM routine a third time. When CONSEM is called the two arguments are passed in the locations NEWCAR and NEWCDR. These two arguments are placed in the CAR and the CDR of the cell returned. The cell is returned by placing a pointer to it in the register CELLPT. The first two calls to CONSEM are followed by steps to make the cells returned look like fixed-point atoms. This is done by setting the CAR's of the cells to point to themselves.

NUMBES -

This SUBR is used to test if the argument passed is a number. It tests if the argument passed is either a floating-point or a fixed-point atom. If the CAR of the argument points to itself, then the argument is a fixed-point atom. NUMBES tests for this condition. If the CAR points to itself the atom T is returned. If not, it tests if the CAR and the CDR of the argument are both fixed-point atoms. If they are, it returns the atom T. If the argument is not a numeric, it returns NIL.

FIXS -

The SUBR FIXS uses a routine in the Interdata double-precision floating-point package to convert a floating-point atom to a fixed-point atom. This routine is named FFIX. FIXS begins by testing to see if the argument passed is a floating-point atom. TSTFLT is used to make the test. TSTFLT stores the floating-point number at FLTNUM. FFIX is called indirectly through the LISP 7/16 internal routine FPT2FI. The routine FPT2FI has two arguments which are passed to it in a parameter block pointed to by register 1. The first element of the parameter block is a pointer to the floating-point number, and the second is a pointer to where the result is to be placed. Since TSTFLT puts the number to be converted in FLTNUM, this is the first parameter. FIXS will call CONSEM to create the fixed-point atom which is the result. Thus the second argument will be NEWCDR. After returning from FPT2FI, FIXS calls CONSEM to get the new cell. After returning from CONSEM the cell is modified to be a fixed-point atom by setting the CAR of the cell to point to itself.

FLOATS -

FLOATS calls the Interdata routine FFLOAT to convert the fixed-point atom passed as the argument into a floating-point argument. The argument passed is tested to be sure that it is a fixed-point argument. FFLOAT requires the value to be passed in register 8 (WORK1) and linkage is via register 15. The result is returned in registers 8 and 9. FLOATS stores this value at FLTNUM and calls the routine CRATFP to get the atom. The result is returned in the register RESULT by CRATFP. This result is then returned by FLOATS.

RPLCDS - RPLCAS -

This pair of routines do RPLACA's and RPLACD's respectively. Each checks the number of arguments making sure that it is correct. The atom NIL is not allowed as the first argument, and these routines check for that condition. The routines also check the address passed to be sure that the location to be modified is within the limits of SPACE. These functions are performed by storing the second argument in the CAR or CDR of the first.

SETS -

This routine checks to see if the first argument passed is a literal atom. If it is, it sets the x'2' bit in the second argument and places it in the CAR of the atom's cell. It returns the value assigned as the result.

INPUTS -

This SUBR checks the argument to be sure it is a fixed-point atom. It then uses the value of the atom to set up the proper function codes for the prompting and reading SVC's. If the device being read from is the Univac 70/46, the prompting is left in, otherwise, it is supressed. This situation is detected because the 70/46 is always assigned to logical unit x'D'. This is done because the 70/46 waits for the 7/16 to ask for more data when sending source programs via the program SRCLOAD. For devices other than the 70/46, it is assumed that they are storage devices, such as the floppy disk or paper tape. In this case prompting should not be given.

OUTPUTS -

This SUBR is like INPUTS except that it is used to change the output device. The argument to the SUBR is the logical unit to which the output is to be sent. This routine also modifies the function code in the proper SVC parameter block.

EOTS -

This SUBR has no arguments and serves to return the input device to logical unit 0. It also turns prompting on by setting the flag.

DEFSUS -

The first argument of this SUBR is tested to be sure that it is an atom. It also checks if the second argument is a fixed-point atom. It uses the SUBR PUTS to place the property SUBR on the property list of the atom with a value equal to the value of the fixed-point atom which is passed as the second argument.

The value of the property SUBR is the address of the routine for performing the SUBR. Linkage to the routine PUTS is done using the standard LISP 7/16 linkage conventions.

PUTS -

This SUBR makes sure that the first argument passed is a literal atom. It then uses the SRCHP routine to search the property list of that atom. In order to link to SRCHP it places a pointer to the property indicator in register WORK2 and the atom in register WORK3. If the indicator is found, the routine returns to the next instruction after the BAL. Otherwise, it returns to the return address plus four. If the property was found, the new value is stored in the property list. SRCHP returns a pointer to the occurence of the property indicator in the property list in the register CELLPT. PUTS uses this result to know where to place the new value. If the property indicator was not found, it is placed at the head of the property list. The structure to be added is formed by two calls to CONSEM. The location NEWCDR is used to save the pointer to the property in case of garbage collection during the first call to CONSEM. After the two new cells are obtained and linked to the old property list, RPLCDS is used to change the CDR of the atom to the new list. The standard LISP 7/16 linkage is used to link to the SUBR RPLCDS.

GETS -

This routine uses SRCHP to see if a value exists. SRCHP returns the value if found in register RESULT. GETS returns the result obtained from SRCHP.

EVALS -

The definition of EVALS is taken directly from the definition of EVAL in the LISP 1.5 Programmer's Manual (McCarthy et. al.). The definition is found on page 71 of that document, and the code follows that definition. The only major differences are that there is no association list either passed as an argument or built (shallow access is used to save variable bindings), and FUNCTION is not supported. The code follows the manual in a very straightforward manner and will not be discussed here. Appendix A contains the definitions of EVAL and APPLY as used in LISP 7/16.

EVCONS -

This routine is a non-recursive version of EVCON as described by the LISP 1.5 Programmer's Manual on page 71. This routine makes sure that the argument is not NIL. It then gets the CAAR of the argument and calls EVALS using the LISP 7/16 linkage conventions to evaluate it. If the result is not NIL, the CDAR of the argument is evaluated, once again using EVALS. If the result of the first evaluation was NIL, the routine loops to LOOPEV after going through EVCDR to get the CDR of the argument. If none of the CAAR's are non-NIL, an error is reported.

EVLISS -

This routine is a non-recursive implementation of EVLIS as described by the LISP 1.5 Programmer's Manual on page 71. It tests if the argument passed is NIL. If it is it returns NIL. If not it goes to the loop which evaluates the CAR of the argument using EVALS. The partial result is CONSed with previous results using CONSEM. The results are saved from garbage collection by making the partial results the new CDR, i.e. storing it at NEWCDR. After using CONSEM the CDR of the cell returned is set to NIL. If this is the first pass through the loop the result is initialized by setting both registers WORK3 and TEMP to point to the cell returned. On succeeding times through the loop, TEMP will be unchanged, but register WORK3 will always indicate where to add the next result returned by EVALS. This loop continues until a NIL is found as the CDR of the argument at which time the whole result is returned.

CONSEM -

This is the internal routine to get cells from the list of free cells. Two arguments are passed to it via the variables NEWCAR and NEWCDR. These arguments will be the CAR and CDR of the cell returned. Register NEXAVL is used exclusively by this routine, and it keeps track of the current available cell. If it is NIL, then garbage collection must take place. If not, NXTAVL is set to the CDR of the next available cell; and the two arguments are placed in the cell being returned. The cell returned is pointed to by register CELLPT, and linkage is via register RTN1.

<u>GABGCL</u> -

This is the internal routine to do garbage collection. It must protect certain lists from the collection. It protects the NEWCAR and the NEWCDR. It also protects the structure pointed to by register TEMP. This is done so that if a SUBR needs to protect a structure it is building, it can use this register to protect it. Also stack bindings are saved, as well as the OBLIST.

The routine begins by saving the registers since they will be needed to do the collection. It then marks the top of the stack by saving its location at TOPSTC. It then gets the OBLIST and marks it using the routine MRKSUB. After it has marked the OBLIST, the collector marks the current structure pointed to by TEMP. The statements between MRKTMP and MARKCA are used to pop up through all the values on the stack and mark passed LAMBDA bindings and old structures pointed to by TEMP.

To see if LAMBDA bindings have been done, the routine adds 16 to the register LSTPSH to see if this equals register NXTPSH. If equal, then no bindings have occurred. If not, then bindings have been made and they must be marked. It also makes a quick test for an error in the stack handling. If no bindings have occurred the stack is popped. If not at the bottom, denoted by NXTPSH value of x'FFFF', the loop continues back to MRKTMP. If LAMBDA bindings have occurred, they are popped from the stack and marked, also using MRKSUB. The value of NXTPSH and LSTPSH are checked again in the loop starting at LAMBLP and ending at MARKCA to be sure that all bindings are marked. After the old stack elements are marked, NEWCAR and NEWCDR are marked using MRKSUB.

The loop starting at DONEMK and continuing to TSTDNE does a linear scan to collect all free cells and convert marked cells back to unmarked cells. Marking cells is accomplished in MRKSUB by setting the lowest order bit of the CAR of cells which are being used. Cells which are collected are linked together to form a list and a count is kept of the number of free cells found. Register NXTAVL is set to point to the first element of the list after printing a garbage collection message and displaying the number of free cells on the front display panel. After garbage collection is done, the collector returns to CONSEM to CONS the NEWCAR and the NEWCDR.

MRKSUB -

This routine is used to mark a list that is passed to it. The list is passed by a pointer in register 3. Linkage to this routine is via register 0. A depth first tree traversal is used to mark the list. The marker descends through the CAR's of a list saving the CDR's on the stack. Descent through the tree stops if the current element is any of the following: NIL, an already marked structure, or a numeric atom. Additionally there are a couple of special conditions which are implementation-dependent and which the marker must detect. One of these is if the current element pointed to is a pname. Recall that the OBLIST is really a list of pointers to pnames and that the OBLIST is one of the structures which must get marked. Thus when a pointer to the pname area is encountered, special handling must be done to follow the list because pnames are not constructed from normal cells. When such a pointer is encountered, the marker must calculate the position of the pointer to the atom and continue its traversal at that pointer. This is done by taking the first byte of the pname entry, which is the length of the ASCII character string, adding two and truncating to a halfword address. This is done because the pointer to the atom is always halfword aligned. The marker then continues using the pointer to the atom as the current structure to mark. If a pointer is out of range of SPACE, then the marker does not follow it. If this happens it is probably an error, but the marker doesn't report it as such. One other special condition is tested for. During reading in LISP 7/16, partial structures are built up and saved on the stack. These structures look like LAMBDA bindings to the marker. The first element points to the head of a partial structure and the second points to where to add new cells. The first element gets marked, but the second element should not be marked. The second element is always a pointer to the CDR of a cell or has the value 1,2 or 3. These constants are used by the reader for the special handling of quotation marks and "dots." These low values are taken care of by the fact that the marker does not follow pointers which are out of the range of space. In the case that the pointer is indicating where to add new structures, it is always a pointer to a CDR of a cell. Thus it must be an odd multiple of two. The marker, looking at such a pointer treats it as if it were the CAR of a literal atom because the x'2' bit is set. The marker thus subtracts two from the pointer and continues the search. If the value was placed by the reader, then subtracting two will leave a pointer to the CAR of the cell. It will always turn out that this is marked already because the first of the two elements on the stack was marked first. When the CAR of a literal atom is really encountered, subtracting two will also yield the correct result. In this case, the CAR of the cell points to the value of the atom plus two. If the atom has no value assigned then the CAR has a value of x'FFFE'; subtracting two will yield x'FFFC'. The marker will not follow this because it is out of the range of space.

Marking is accomplished by setting the lowest order bit in the CAR of the cell. Only previously marked cells should have this bit set. The conditions mentioned above are tested in the following order:

1.) If NIL then pop a CDR if there is one.

2.) If inside the pname area calculate the address of the pointer to the atom and follow it.

3.) If above SPACE then skip this item.

4.) If an odd multiple of two then subtract two and follow it.

5.) If previously marked then pop a CDR if there is one.

If none of these special conditions are met then the cell is marked, the CDR is pushed onto the stack, and the marker follows the CAR unless the cell is numeric, in which case a CDR is popped. Since a floating-point atom is really a dotted pair of fixed-point atoms, no special handling is needed for floating-point atoms.

When a CDR is popped from the stack, the marker tests to see if it is empty. If it is then the marker returns. If not then it puts the top stack element in register 3, decrements the top of stack pointer and goes back to the code to do the marking.

<u>READS</u> -

This is the largest routine in LISP 7/16. The basic strategy is to build up structures using the stack to hold partial results. The occurrence of a left parenthesis in the input signals the need to introduce a new stack element. Right parentheses indicate a stack element should be popped and returned to a higher level. Two halfwords are pushed onto the stack whenever a push is done. The first of these will point to the structure being built and the second will point to where to add new elements to the structure. The second element is also used to mark the occurrence of the special characters, quotation mark (apostrophe) and dot (period). This routine has no arguments. It begins by marking the top of the stack by saving a pointer to it at TOPSTK. The result is tentatively set to NIL, and checking is done to see if prompting should be done. If reading from a storage device then prompting should not be done. The prompting is done using the special MOSS SVC which does not follow every line of text with a carriage return and line feed, as BOSS SVC's normally do. Whether prompting is done or not, the routine then reads a line of text from the input device. A dummy wait loop has been added for communication with the Univac 70/46.[1] After reading a record, the prompting character is modified to indicate that continued input is being read. The routine also tests if it was called from the loop EVALOP which is the basic evaluation loop. A different prompting character is used if this is not the case.

After reading the line of text, leading blanks are skipped over. Register TEMP is used to point to the current character in the input buffer and the code at INCTMP is used to update this value when scanning over the buffer. If a non-blank character is found, the reader tests for leading ASCII null characters. These may occur in reading from the 70/46. The current character is tested to see if it is a comment character. If it is then control transfer to a section of code called TSTND1 which tests if the reader is at the top of the stack. If it is, then the reader will return, otherwise; it will request more data. READS next checks for the occurrence of a left parenthesis, a right parenthesis, a dot or a quotation mark. If one of these are found then the reader will transfer to LEFTP, RIGHTP, DOT, or QUOT respectively. Each of these will be described later. If the current character is none of these special characters, the reader tests if the current character is a letter. If the current character is less then the character 'A' (in ASCII representation) then it may be a number and that possibility is checked at MAKNUM. If the character is greater then 'Z' then a syntax error has occurred, and the error is reported. If none of the branches above have occurred then the current character should be the start of the pname of some literal atom. Two internal routines are called upon to operate on it. The first of these, GETNAM, will extract the name from the input stream, check to make sure it is valid, and place it at the location PNAM with its length

1. The Univac 70/46 communications require that the communications hardware has enough time to turn the line around.

prefixing it. The second is FNDPNM which will test to see if the atom exists already. If it does, then a pointer to it will be returned; if not, then an atom and pname entry will be created and the pointer to it returned. The reader will then branch to RETSTR which is the code used to build the structures.

GETNAM searches the input stream using the value in TEMP as a pointer to the current character. The scan continues until a delimiter is found. The delimiter may be a parenthesis, a dot, a space, a carriage return, or a non-alphanumeric character. The scan may also end if twenty characters have been read since this is the limit of the number of characters allowed. If more than twenty characters are found then an error is reported. The characters in the input string are moved into the area starting at PNAM+1 as they are scanned.

FNDPNM begins by storing the length of the pname at PNAM in the first byte. The four low order bits of the first character of the pname are then used to hash into the OBLIST. The low order bits must be multiplied by four to get the proper cell in the OBLIST since each entry is four bytes long. The label OBHASH points to the head of the hash list and it is used in conjunction with the hashing value to get the correct list. The code at HSHLP to SAVPNT is a linear search through the list. If the list is NIL then the pname contained at PNAM is not in the OBLIST, and control transfers to NOTIN. If not NIL, then the length of the pname in question is compared to that of the first element in the list. If equal, then the characters are compared one by one by the code at the label NAMLP. If all the characters are the same then the pname already exists and control passes to INTHER. If not, then the process is repeated with the CDR of the list. Before going back to look at the remainder of the list, the current element of the list is saved at SAVPNT. This is done because the next element may be NIL which means a new element of the list must be added and SAVPNT will tell where that element should be added.

If the pname was found then the code at INTHER will get the pointer to the atom and load it into register RESULT to return it as the result. If the pname was not there then the routine MAKEAT is used to create the atom. MAKEAT must move the pname to the pname area at the next available location, get a cell, mark it as an atom with no value assigned, place the pointer to it in the pname area, add it to the hash list and return a pointer to the atom.

The variable TOPOB points to the next available space in the pname area. MAKEAT calculates the length of the entry it will make and then checks if room is left in the pname area. The length of the entry is the length of the ASCII string plus one for the length byte, plus two for the pointer to the atom, plus one, rounded to a halfword address. The extra one is added in case padding is needed to align the atom pointer on a halfword boundary. If there is enough space for the new entry, the pointer TOPOB is updated. The new literal atom cell is created with a NIL CDR (the property list pointer) and a CAR of x'FFFE' (no value assigned). This new cell is acquired using CONSEM. This cell is the result, and so a pointer to it is stored in RESULT. The loop at BYTELP moves the characters one at a time into the proper place in the pname area. Another cell is acquired using CONSEM and this cell is placed in the OBLIST by using the address saved at SAVPNT. In the special case of an initially empty hash list, the pointer to the pname entry is saved directly in the top level of the OBLIST. Otherwise, it is inserted as the CDR of the last element of the old hash bucket.

In case of a quotation mark in the input, special handling is required. This mark implies a set of parentheses. If a quotation mark is found, a new stack element is created. The second halfword of the stack element is set to x'3' to mark the occurrence of the quote. The first halfword is set to point to the structure being built. In this case, the CAR of that structure should be the atom QUOTE. The CDR of the structure will be a pointer to the structure. That structure has not been read yet. CONSEM is used to begin the construction of the structure. A cell is obtained and the CAR of it is set to point to QUOTE. After doing this, control passes to INCTMP to continue scanning the input buffer.

If an open parenthesis is found then the code at OPENP is executed. It creates a null stack element with both elements set to zero. These elements will be filled in by RETSTR when the elements inside the parentheses are scanned and returned.

If a dot is found then the code at DOT will be executed. This code will first check to see if the reader is at the top of the stack. This is a syntax error since a dotted expression must appear inside a pair of parentheses. If the first element of the stack is zero then this is also a syntax error since something must precede the dot. The routine checks to see if the second halfword of the stack element is a pointer to the CDR of the cell pointed to by the first halfword. This is the proper set of circumstances since it means that only one element has preceeded the dot. If this is not the case either several elements have preceeded the dot (eg. (A B . C)) or more than one dot appeared at the current level (eg. (A..B) or (A.B.C)). If the occurrence of the dot is not a syntax error then the stack is marked by placing a x'1' in the second halfword of the current stack element. This indicates to RETSTR that a dot has occurred.

RETSTR takes the structure passed in register RESULT and patches it into the structure pointed to by the top stack element. RETSTR first tests if a quote has occurred. If one has then it uses CONSEM to build the list consisting of QUOTE and the structure being returned. A new cell is acquired with CDR of NIL and CAR the structure being returned. This new cell is patched in using the information stored on the stack. In this case, the first halfword points to the CAR of the cell whose CDR must be set to the new cell acquired. After the patching is done, control passes to CLOSEP since a quote implies a pair of parentheses in the input. If not closing a quote, RETSTR tests if is completing a dotted pair. If it is, then the second halfword of the top stack element is set to x'2', which prevents multiple dots at the same level. The routine also makes a test to be sure that the structure was not (.) which is not allowed. Control passes to INCTMP to continue scanning. RETSTR checks to be sure that two dots have not occurred by testing to see if the second halfword of the top stack element is a x'2'.

If the second halfword of the top stack element is not x'1', x'2', or x'3', then a normal list is being built. In this case, RETSTR obtains a new cell using CONSEM. The CAR of this new cell is set to the structure being returned, and the CDR is set to NIL. The cell is then patched into the structure being read by using the second halfword on the stack. The new cell is placed at the position that this halfword indicates. A pointer to the CDR of the new cell is placed as the second element of the stack. The routine then passes to INCTMP to continue the scan of the input buffer.

When a closing parenthesis is found, the stack is popped. If there is no element to pop, a syntax error has occurred. The code at CLOSEP does the popping. It checks to make sure that if a dot has occurred there is an expression between it and the closing parenthesis. The absence of such an expression is indicated by a x'1' in the second halfword of the top stack element. The first halfword of the top stack element is the result, and it is returned in register RESULT. If the stack is empty then the READ routine is exited. If not, then the result is returned via RETSTR to the next higher level. Before leaving in the case of an empty stack, the routine checks to make sure there are no unscanned characters in the input buffer other than comment characters. If there is, then an error is reported.

MAKNUM is used to create numeric atoms when needed. The string is tested to see if it starts with a number between 0 and 9 or a plus or minus sign. If not then a syntax error is reported. Otherwise, the routine ASC2BI is used to get the number. Upon return the character that follows the number found is examined. This is done because the routine which actually converts the characters to numbers reads up to six characters and stops. If the next character is a decimal point or another number, then the number was a floating-point atom. In this case control passes to FLTER. If the number was not a floating-point number, a new cell is acquired using CONSEM; and it is converted into a fixed-point atom by setting the CAR to point to itself. This cell is returned to RETSTR. If the number is floating-point then the routine ASF2BI is used to get the number. This routine will place the result at FLTNUM. The routine CRATFP is then used to create a floating-point atom. The result is returned by CRATFP in register RESULT, and this is returned to RETSTR. Both ASF2BI and ASI2BI properly adjust TEMP so that it points to the next character.

GENSYS -

This routine begins by using the routine SIBTOD which is supplied by Interdata. It is used to convert the value stored at NXTSYM to a printable character string. After doing this the value of NXTSYM is incremented so that the next call to GENSYM will create a new atom. The length of the pname of the atom and the character 'G' are prefixed to the ASCII string. FNDPNM is then used to create the atom and place it on the OBLIST. FNDPNM returns the result in register RESULT and GENSYS passes it along as the result.

CARS - CDRS - CONSS -

These three SUBR's use the internal forms GCARW1, GCDRW1 and CONSEM, respectively, to perform the operations.

EQS -

This routine tests if the two arguments passed are equal, i.e. have the same value. If they are the same it returns the atom T; otherwise, it returns the atom NIL.

ATOMS -

This routine begins by testing if the argument passed is a fixed-point atom. It then uses TSTFLT to test if it is a floating-point atom. Finally it tests if the x'2' bit is set to indicate it is a literal atom.

SPREAD -

This routine is used by APPLY to place pointers to arguments in the registers WORK1, WORK2 and WORK3. It can take at most three arguments. It counts the number of arguments in the list and leaves the count in the register NUMARG for checking by SUBR's.

PRINTS -

This routine prints its arguments and returns the argument passed as the result. The SUBR begins by marking the top of the stack. PRINTS will use the stack in doing a print of the structure passed. PRINTS is essentially a recursive routine. If the argument passed to it is an atom, it will print it and return. If the argument is not an atom, it will push the CDR of the structure on the stack and print the CAR. After printing the CAR, it will pop the CDR and repeat the process. PRINTS will try to print its argument as a list. If it cannot (because some S-expressions are not lists), this means the last element of the structure is not NIL. In this case it will print a dot ('.') followed by the last element of the structure. Thus (A B . C) will be printed for the structure (A . (B . C)).

PRINTS marks the top of the stack by saving a pointer to it at SAVARE. It will use register RESULT to point to pnames of the atoms when it must pass them as parameters to the subprogram MVPNAM which will move them into the print buffer. Register CHAR is used to hold a character to be moved into the print buffer when calling the subprogram MVCHAR. The register TEMP is used to hold the number of unmatched left parentheses in the output. This count is used in a "pretty print" routine built into PRINTS.

PRINTS begins by marking the stack, initializing the count of unmatched parentheses to zero, initializing the length of the item to be printed to zero, and saving the argument passed so that it might return it as the result. PRINTS also places the CAR and the CDR of its argument, which is passed in WORK1, into the registers WORK2 and WORK3, respectively.

At the label PRNTAT, PRINTS tests if its argument is an atom, either NIL, fixed-point, floating-point, or literal. If it is none of these then PRINTS transfers to the label NOTATM. Since its argument is not an atom it must be a structure. Therefore a '(' must be printed. The routine MVCHAR is used to move this character into the print buffer. The argument passed, its CAR and its CDR are saved on the stack; and control passes back to PRNTAT which again begins by checking for an atomic argument. If an argument passed is an atom then the appropriate print name is moved into the print buffer. If a fixed-point atom is detected then the subprogram MOVNUM is used. If a floating-point atom is detected then MOVFLT is used. If NIL is passed as the argument then a pointer to the print name of NIL is passed to the subprogram MVPNAM. If any other literal atom is detected then the subprogram FNDNAM is used to find its pname, and the result is then passed to MVPNAM. MOVNUM, MOVFLT and MVPNAM will move the appropriate ASCII characters into the print buffer. After doing so, control will pass to POPCDR. The code at POPCDR tests if the stack is empty. If so, printing is complete; and PRINTS returns. If the stack is not empty a CDR is taken off the stack, and control passes to RETURN.

RETURN tests if the CDR popped from the stack is NIL. If it is, then it marks the end of a list. In this case a ')' is moved into the print buffer using MVCHAR, and control passes to POPCDR to pop the stack again. If the CDR passed to RETURN is atomic then it marks the end of a structure which is not a list. Therefore a dot must be printed followed by the atom and a ')'. As before, checking is done to test for a numeric or literal atom. Also, as before, the [illegible]

MVPNAM, MOVNUM, and MOVFLT are used to move the appropriate characters into the print buffer. If the CDR passed to RETURN is not atomic then a pointer to it is loaded into register WORK1, its CAR and CDR are placed in WORK2 and WORK3 and control is passed to PRNTAT once again. This is done at the label SETQS.

The subprogram MVCHAR is used to move single characters into the print buffer. Linkage is via register RTN1. The subprogram MVCHAR counts the number of left parentheses it has moved into the print buffer. If it is passed a '(' it will flush the existing buffer using PRNTIT and then place the character into the buffer. The print buffer contains a leading byte which defines the length of the string in the buffer. MVCHAR uses this byte in the code following NOTRP to calculate where to place the character. MVCHAR decrements TEMP, the count of unmatched left parentheses, whenever it is passed a right parenthesis. In the case of a right parenthesis, it does not print the buffer. When inserting a right parenthesis into the print buffer, MVCHAR will delete spaces preceding it. This deletion is done with the code beginning at INSERT and ending at TSTLNG. The code at TSTLNG tests if the print buffer is full. If so then it calls PRNTIT to print the buffer. Otherwise, it returns to the place from which it was called.

The subprogram PRNTIT is used to print the buffer. It saves the linkage register, RTN2, at SAVREA+6, since RTN2 will be used by PRNTIT to link to another subprogram, INDENT. Before printing the buffer, a tape-off (ASCII character) and a carriage return are inserted into the buffer. The tape-off is inserted in case the output is going to the 70/46. The carriage return is used to mark the end of the buffer for the BOSS SVC's. After printing the buffer, PRNTIT calls the subprogram INDENT in order to indent the proper amount for pretty printings.

MVPNAM is used to move a print name into the print buffer. Linkage to MVPNAM is via register RTN1. RESULT points to the print name to be moved. MVPNAM first checks if there is room left in the buffer. If not, it is printed using PRNTIT. The first byte of the buffer is used to calculate the length of the resulting buffer after the print name is moved in. The loop at PNLOOP is used to move the characters into the buffer beginning with the last. After all are moved in MVPNAM returns.

INDENT is an internal routine to indent the proper amount at the start of each line to be printed. It is linked to via the register RTN2. The count of unmatched parentheses is used to do this. The number in TEMP is tested to see if it is zero. If so then no indentation is needed and control passes to INBLN which sets the length of the buffer to zero in this case. If TEMP is not zero then it is decremented by one and multiplied by two. The result is the number of spaces to indent. The loop at INLOOP actually moves the spaces into the buffer.

MOVNUM is used to move fixed-point numbers into the buffer. It uses the routine BI2ASI to convert the number into the appropriate ASCII character string. BI2ASI is passed its parameters in a block pointed to by register 1. The parameters are the address of the number to convert and the address of where to place the result. MOVNUM places the number at FIXNUM, and the result will be found at PNUMBR. The parameter block is located at CNVRT. The length of a fixed-point print name is always six, and this length is stored at PNLENT+1. The data stored at PNLENT+1 and PNUMBR thus looks like a print name. MVPNAM is called to move it into the print buffer. RESULT is loaded with a pointer to PNLENT+1 before the call to MVPNAM. After moving the print name, control passes back via the link register.

MOVFLT also uses the link register RTN2. It is very similar to MOVNUM, and it shares the code at RETNUM inside MOVNUM. MOVFLT will first set up a parameter block for BI2ASF. This parameter block is passed in register 1. The parameter block contains the address of the number and the address for the result. Before arriving at MOVFLT, a call to TSTFLT must have been made. This routine leaves the floating-point number that it found at FLTNUM. MOVFLT uses this fact to [illegible] the number to BI2ASF. After calling BI2ASF, control passes to the common code at RETNUM. BI2ASF returns a pointer to the last byte of the result it constructed. MOVFLT uses this to calculate the length of the result. RETNUM will store this length in the result.

<u>SRCHP</u> -

SRCHP is an internal routine to search a property list. It is passed arguments in registers WORK2 and WORK3. WORK2 points to the property indicator and WORK3 points to the atom whose property list is to be searched. RESULT will return NIL upon return if the indicator was not found. It will return a pointer to the value associated with the indicator if it is found. CELLPT returns a pointer to the occurrence of the indicator in the property list. WORK2 and WORK3 are not changed. Linkage is via register RTN1. If the indicator is found on the property list SRCHP returns to the location in RTN1. Otherwise, it returns to the location in RTN1 + 4. SRCHP checks to make sure that the property list is of even length. If not, then an error is reported.

<u>GCARW1</u> - <u>GCARW2</u> - <u>GCARW3</u> -

These routines are used to get the CAR of register WORK1, WORK2 and WORK3 respectively. Linkage is via register RTN2. The result is returned in register RESULT. If the argument passed is a literal atom then these routines will return the value of the atom unless it is undefined in which case an error will be reported.

<u>GCDRW1</u> - <u>GCDRW2</u> - <u>GCDRW3</u> -

These routines are similar to the above routines except that they get the CDR of the argument.

<u>APPLYS</u> -

The SUBR APPLYS is a machine language version of APPLY as given on page 70 of the <u>LISP</u> <u>1.5</u> <u>Programmer's</u> <u>Manual</u>. The major differences are the elimination of FUNARG, LABEL and the association list. Appendix A gives a definition of APPLYS.

PAUSES -

This routine uses the BOSS SVC to cause a break to the operating system.

TSTFLT -

This routine is used to test if a cell is a floating-point atom. Upon calling the routine, WORK1 should point to the cell; and WORK2 and WORK3 should point to the CAR and the CDR of the cell, respectively. The routine returns the atom T if the cell is a floating-point cell and NIL otherwise. The floating-point number is stored at FLTNUM if the cell is a floating-point atom.

BI2ASI -

This routine is used to convert binary numbers to ASCII integers. It links to the Interdata routine, SIBTOD.

ASI2BI -

This routine reads up to six characters from the input buffer and returns the proper binary integer. This routine begins by saving the registers 10 through 15. The parameters are passed to the routine in a parameter block which is pointed to by register 1. The first parameter is the address of the place to return the result. The second parameter is the address of the ASCII character to be converted. Register 10 is used as a pointer to the binary and register 11 as a pointer to the ASCII string. Register 12 is used to hold the value of the result. Register 13 is used to hold the current character extracted from the buffer. Register 14 is a temporary used in multiplication, and register 15 is a count of the number of characters read. The routine initializes some registers and looks for a sign. If a plus sign is found, it is skipped over. If a negative sign is found, the flag NEGFLG is set; and conversion continues. The loop [illegible] is used to extract the ASCII characters from the buffer. They are checked to see if they are numeric. If not, the end of the string is assumed. Each character is stripped of high bits, and the previous result is multiplied by ten using two shifts and an add. The new value is then added in. When the loop is exited, either a non-numeric character has been found or five digits have been extracted. After leaving the loop, NEGFLG is tested to see if the complement is needed. If so, the result is complemented. The result is stored at the proper address, and the address of the last ASCII character used is returned in the parameter block.

FPT2FI - ASF2BI - BI2ASF -

These routines are used to link to the Interdata routines, FFIX, FDBCNV and FBDCNV, respectively. They fix a floating-point number, convert an ASCII floating-point number to binary and convert a binary floating-point number to ASCII respectively. FPT2BI has two parameters which are passed in a parameter block via register 1. The first parameter is the address of the floating-point number and the second is the destination for the result. ASF2BI and BI2ASF have two parameters also. The first is the address of the binary number and the second is the address of the ASCII number.

PUSH -

This routine is called by all SUBR's upon entry. It saves registers 8 through 15. It tests for stack overflow and clears register TEMP. TEMP is cleared since the garbage collector preserves structures pointed to by TEMP, and therefore clearing it helps prevent the inadvertent protection of garbage.

POP -

This routine is branched to by each SUBR to return. It pops the stack, checking for LAMBDA bindings. If any are found, they are released and old values are restored. The bindings on the stack consist of two halfwords. The first of these is a pointer to the atom and the second a pointer to the old value plus two. The plus two is there so that when the value is restored to be the CAR of the atom, it will look like a literal atom. LAMBDA bindings are detected through the use of the forward and backward pointers stored on the stack. If they differ by sixteen then no bindings have occurred as this is the length of the push [illegible] on entry to a SUBR.

RECOVR - NCOVER -

These routines and the short sections of code before them handle errors. Each of the error routinés loads an error code into register LEN and passes to one of these routines. RECOVR is used for errors which are recoverable. NCOVER is used for more serious errors which may not always be recovered from. RECOVR prints an error message and moves the error code passed in register LEN into the error message. The routine then prints the atom pointed to by the register ATMPNT. This register should contain a pointer to the function being executed when the error occurred. EVAL and APPLY load this register when interpreting a SUBR, EXPR, FSUBR, or FEXPR. All SUBR's except PRINT store the number of parameters passed at ARGNUM. This is done so that in case of an error, the arguments may be printed. PRINTS is called to print the arguments; thus it is not allowed to modify this number. After printing the arguments the routine does a PAUSE to the operating system. If the user continues, the program branches to RESTRT. This does not free LAMBDA bindings, but the stack is effectively popped. NCOVER acts similarly except it only prints the error message and then pauses.

```
eval[form]=[
   null[form]→NIL;
   numberp[form]→form;
   atom[form]→[get[form;APVAL]→apval;*
      T→[eq[car[form];QUOTE]→cadr[form];
         eq[car[form];COND]→evcon[cdr[form]];
         eq[car[form];PROG]→prog**[cdr[form]];
         atom[car[form]]→[
            get[car[form];EXPR]→
               apply[expr*;evlis[cdr[form]]];
            get[car[form];FEXPR]→
               apply[fexpr*;list[cdr[form]]];
            get[car[form];SUBR]→
              {spreard[evlis[cdr[form]]];
              {BAL RTNADR,subr*
            get[car[form];FSUBR]→
              {WORK1:=cdr[form];
              {BAL RTNADR,fsubr*
            T→eval[cons[value[car[form];
                  error[VA]];cdr[form]]]];
         T→apply[ear[form];evlis[cdr[form]]]]];
```

value[X;Y]

The function value retrieves the value of the first argument X which must be a literal atom. If X has no associated value then the form Y is evaluated.

error[indicator]

The function error prints the indicator passed as an argument.

```
apply[fn;args]=[
   null[fn]→NIL
   atom[fn]→[get[fn;EXPR]→apply[expr;args];
             get[fn;SUBR]→ spread args ;
                           BAL RTNADR,subr*
             T→apply[value[fn;error[VA]];args]]
   eq[car[fn];LAMBDA]→bind[cadr[fn];args];
                       eval[caddr[fn]];
   T→apply[eval[fn];args]];
```

bind[var-list;val-list]

The function bind take two lists of equal length as arguments. The first list must be a list of literal non-NIL atoms. The values in the second list are bound to the first list in the order they appear.

* The value of get is set aside. This is the meaning of the apparent free variables.

** Although prog appears, it is not supported. The subr for prog simply reports an error message.

Appendix B:

Register Convention and Linkage Conventions

LISP-716

Register Conventions

NUMBER	NAME	USE
0	RTN1	General purpose linkage register for internal subroutine calls. It may be used as a temporary.
1	NUMARG	Holds the number of arguments passed to a SUBR. It is set up by the routine SPREAD. It may be used as a temporary within a SUBR.
1	RTN2	General purpose linkage register for internal subroutine calls.
2	CELLPT	Returns a pointer to the cell returned when requesting a cell by using CONSEM. It may be used as a temporary.
3	RESULT	Used to return results when exiting. It may be used as a temporary.
4	RTN3	General purpose linkage register. Used to return and error code to the recover routine. It may be used as a temporary.
4	LEN	It is used within READ and PRINT to hold the length of a pname.
5	CHAR	May be used as a temporary. It is used by READ and PRINT to hold characters.
5	ARGPNT	Used by EVAL to point to argument it was passed.
6	NXTAVL	Holds address of next available cell in free space - not to be used or changed by subrs!
7	SUBADR	Holds the address of the SUBR. May be used as a temporary.
8	WORK1	Holds address of the first argument passed a SUBR.
9	WORK2	Holds address of the second argument passed a SUBR.
10	WORK3	Holds address of the third argument passed a SUBR.
11	TEMP	A general temporary which can be used to save a structure from garbage collection during a call to CONSEM.

NUMBER	NAME	USE
12	NXTPSH	Address of next element in the stack. The value passed in this register must be returned in it when exiting a SUBR. A SUBR may use the stack locations starting at the one pointed to by this register, but on exit the value should be restored to original value passed.
13	ATMPNT	Pointer to the atom associated with the SUBR being executed. It is used in error printing and may be modified after all error checking has been done in the SUBR.
14	RTNADR	Return address for the SUBR. POP uses it to return. The user may use it as a temporary after a call to PUSH which saves it.
15	LSTPSH	Address of the last element on stack. Should not be changed.

Linkage Conventions

Upon entering a SUBR the following instructions should be executed to maintain compatability with LISP 7/16 SUBRS:

```
NAME    BAL     RTN1,PUSH           PUSH A STACK ELEMENT
        STH     NUMARG,ARGNUM       SAVE NUMBER OF ARGUMENTS
        CLHI    NUMARG,NUMBR        TEST FOR PROPER NUMBER
        BNE     ERROR               GO TO ERROR ROUTINE
```

PUSH and ARGNUM are entry points in LISP- 7/16 and should be EXTRNS to the SUBR being executed if it is to be linked to LISP. The variable NUMBR is a local variable which is the number of arguments expected. Errors are handled by loading the register LEN with a two character ASCII error code which will be printed. The SUBR should then branch to RECOVR or NCOVER depending on whether the error is fatal or not. Upon leaving the SUBR, the register RESULT should be loaded with the result, and the SUBR should branch to POP. POP, RECOVR, and NCOVER are entry points in LISP 7/16 and should be EXTRNS in the SUBR if is to be linked to LISP. If a SUBR is not going to call any other SUBRS and there are no recursive calls to itself, then the call to PUSH may be eliminated. The SUBR can then return via the link register RTNADR.

Appendix C:

Illustrations at ATOMS, the OBLIST, property list, and the stack

Fig. 1a.   Structure of literal atoms

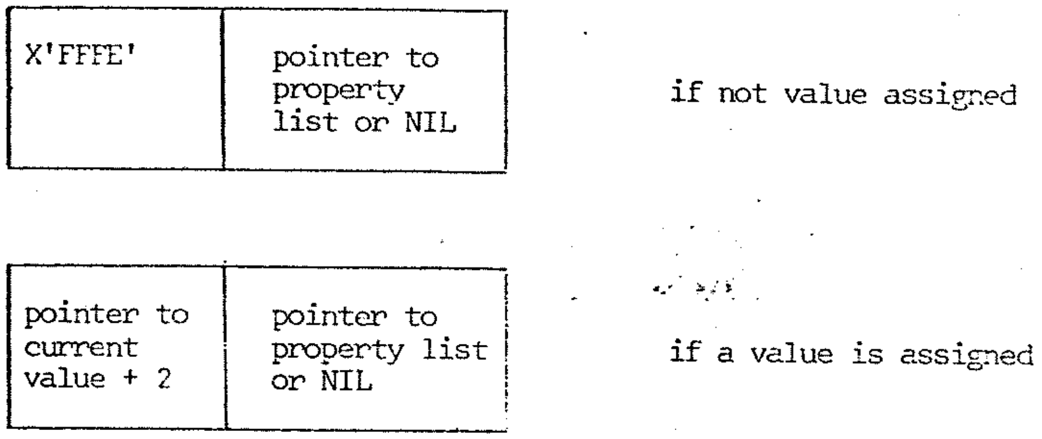

X'FFFE'
pointer to property list or NIL
if not value assigned
pointer to current value + 2
pointer to property list or NIL
if a value is assigned

Fig. 1b.   Structure of fixed point atoms

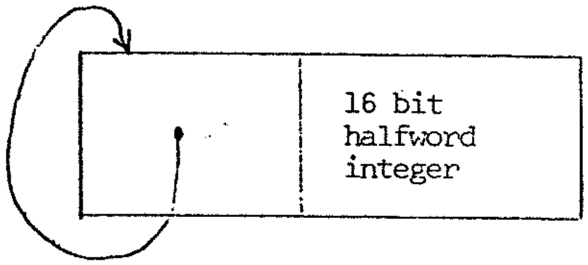
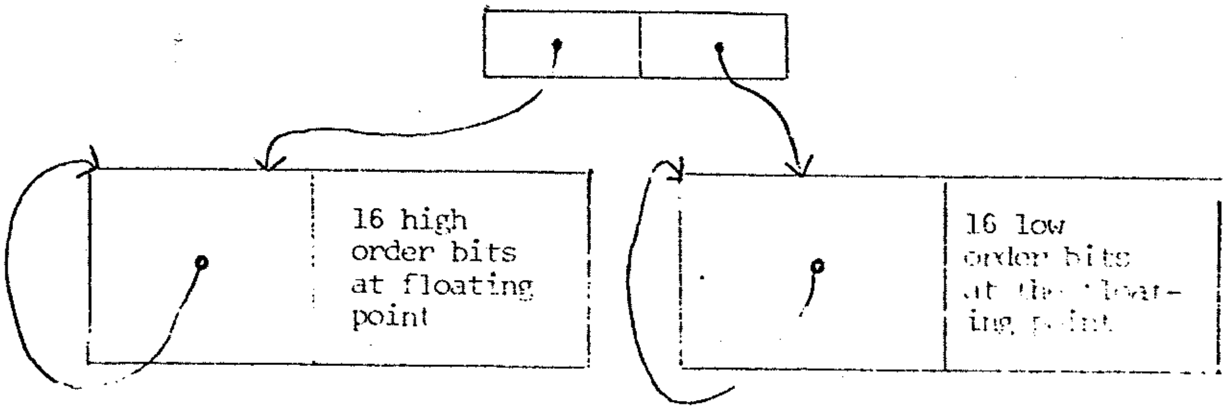
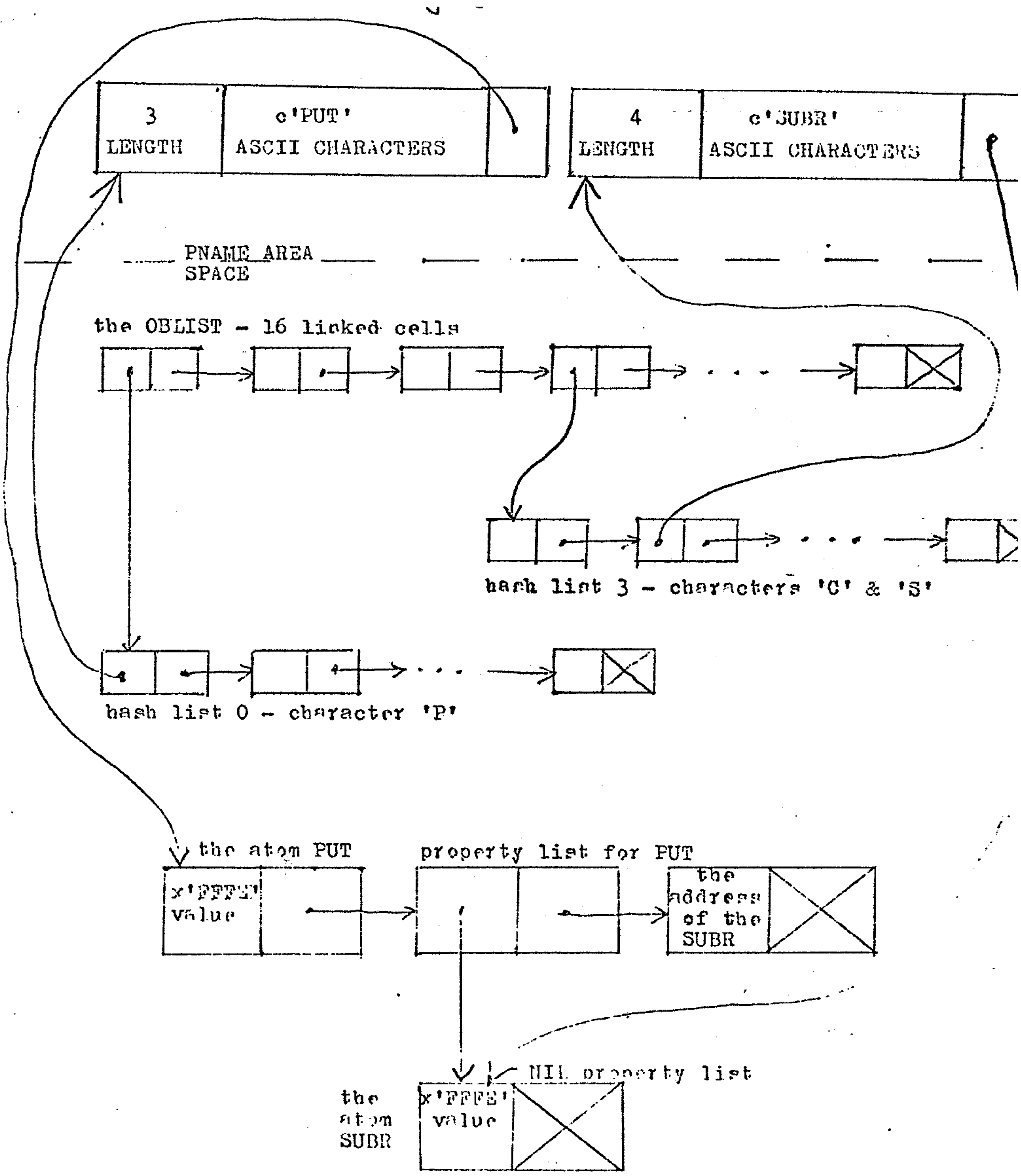


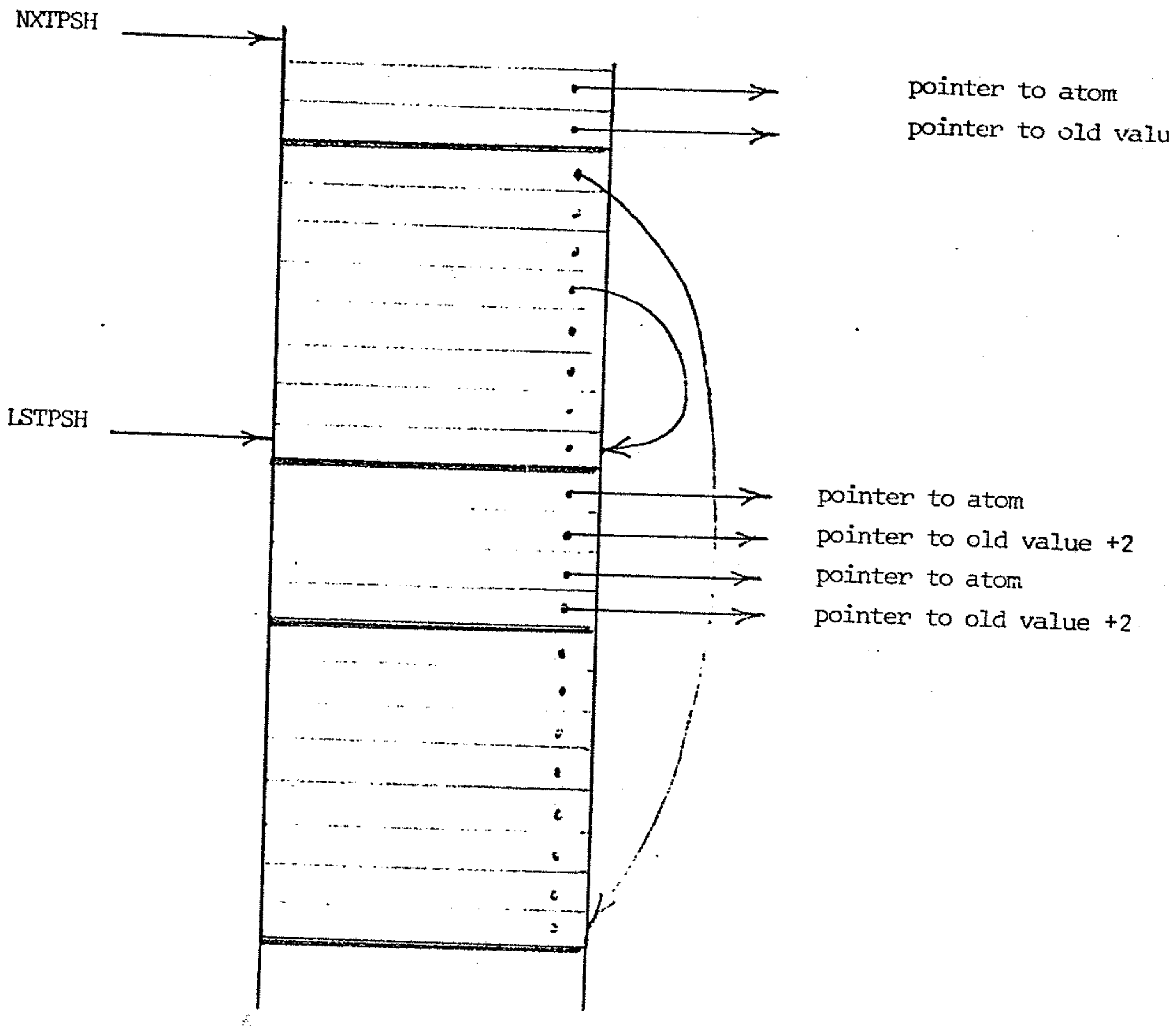
Fig. 1c.   Structures of floatingpoint atoms

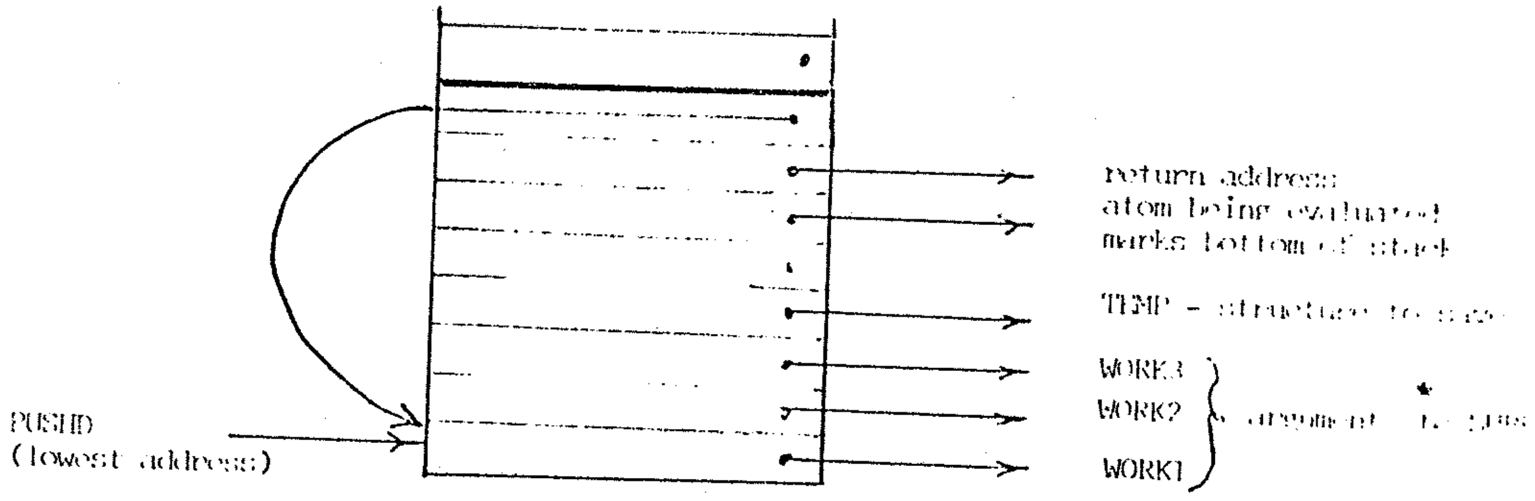

16 high order bits at floating point
16 low order bits at the floating point



This figure illustrates the OBLIST, [illegible], property lists and their relati
It shows the atoms 'PUT' and 'SUBR'. The property list of SUBR is NIL. Th
property list of PUT has a pointer to SUBR and the address of its subr.

The Stack


NXTPSH
pointer to atom
pointer to old valu
LSTPSH
pointer to atom
pointer to old value +2
pointer to atom
pointer to old value +2


return address
atom being evaluated
marks bottom of stack
TEMP - structure to save
WORK3
WORK2
WORK1
argument [illegible]
PUSHD
(lowest address)

MODULE CLASSIFICATION AND LINKAGE

NAME	TYPE[1]	LINKAGE	ARGUMENTS/COMMENTS
INIT	interp.		Initializes data areas
RESTRT	interp.		Restart address
EVALOP	interp.		Basic interpretive cycle
ADDS	SUBR	standard[2]	LISP SUBR : (ADD X Y)
SUBS	SUBR	standard	LISP SUBR : (SUB X Y)
MULTS	SUBR	standard	LISP SUBR : (MULT X Y)
DIVIDS	SUBR	standard	LISP SUBR : (DIVIDE X Y)
LESSS	SUBR	standard	LISP SUBR : (LESS X Y)
GREATS	SUBR	standard	LISP SUBR : (GREATER X Y)
EQUALS	SUBR	standard	LISP SUBR : (EQUAL X Y)
ARITHS	internal	RTN3	WORK1 and WORK2 / uses RTN3 only for compare, otherwise, returns using POP.
CRAFFP	internal	RTN2	FLTNUM must hold number. Returns result in RESULT.
NUMBES	SUBR	standard	LISP SUBR : (NUMBER X)
FIXS	SUBR	standard	LISP SUBR : (FIX X)
FLOATS	SUBR	standard	LISP SUBR : (FLOAT X)
RPLCDS	SUBR	standard	LISP SUBR : (RPLACD X Y)
RPLCAS	SUBR	standard	LISP SUBR : (RPLACA X Y)
SETS	SUBR	standard	LISP SUBR : (SET X Y)
INPUTS	SUBR	standard	LISP SUBR : (INPUT X) / read from logical unit
OUTPUS	SUBR	standard	LISP SUBR : (OUTPUT X) / write to logical unit
EOTS	SUBR	standard	LISP SUBR : (EOT) / end of transmission from this logical unit; read from logical unit 0..
DEFSUS	SUBR	standard	LISP SUBR : (DEFSUBR X Y) / Define X as a [illegible] with location Y.
PUTS	SUBR	standard	LISP SUBR : (PUT X Y Z)
GETS	SUBR	standard	LISP SUBR : (GET X Y)
EVALS	SUBR	standard	LISP SUBR : (EVAL X)
EVCONS	SUBR	standard	LISP SUBR : (EVCON X)
EVLISS	SUBR	standard	LISP SUBR : (EVLIS X)
CONS[illegible]	internal	RTN1	NEWCAR and NEWCDR. Result returned in [illegible]T. Does garbage collection if needed. Protects TEMP during garbage collection

TABLE 1. (continued)

NAME	TYPE	LINKAGE	ARGUMENT/COMMENTS
GABGCL	internal		Does garbage collection.
MRKSUB	internal	RTN1	Register 3. Marks structure passed.
READS	SUBR	standard	LISP SUBR : (READ)
GENSYS	SUBR	standard	LISP SUBR : (GENSYM)
CARS	SUBR	standard	LISP SUBR : (CAR X)
CDRS	SUBR	standard	LISP SUBR : (CDR X)
CONS	SUBR	standard	LISP SUBR : (CONS X)
EQS	SUBR	standard	LISP SUBR : (EQ X Y)
ATOMS	SUBR	standard	LISP SUBR : (ATOM X)
SPREAD	internal	RTN1	RESULT points to list to SPREAD. WORK1, WORK2 and WORK3 return arguments. NUMARG returns a count of the arguments.
PRINTS	SUBR	standard	LISP SUBR : (PRINT X)
SRCHP	internal	RTN1	WORK2 points to property. WORK3 points to the atom. RESULT returns result. CELLPT points to occurence of indicator in the property list. Returns to RTN1+4 if not found. / Searches the property list looking for a property.
GCARW1	internal	RTN2	Gets CAR of WORK1.
GCARW2	"	"	" " " WORK2.
GCARW3	"	"	" " " WORK3.
GCDRW1	"	"	" CDR " WORK1.
GCDRW2	"	"	" " " WORK2.
GCDRW3	"	"	" " " WORK3.
APPLYS	SUBR	standard	LISP SUBR : (APPLY X Y)
PAUSES	SUBR	standard	LISP SUBR : (PAUSE) / Pause to the operatin[illegible] system.
TSTFLT	internal	RTN1	WORK1 points to the cell. WORK2 and [illegible] CAR and CDR of the cell. Number returned in FLTNUM.
BI2ASI	internal	RTN1	Parameter block in register 1. Address of bin[illegible] where to place ASCII.
ASI2BI	internal	RTN1	Parameter block in register 1. Address to plac[illegible] binary and address of ASCII.
FPT2FI	internal	RTN1	Parameter block in register 1. Address of flo[illegible] ing and where to place fixed.

TABLE 1. (continued)

NAME	TYPE	LINKAGE	ARGUMENTS/COMMENTS
ASF2BI	internal	RTN1	Parameter block in register 1. Address of ASCII and where to place binary.
BI2ASF	internal	RTN1	Parameter block in register 1. Address to place ASCII and address of binary.
PUSH	internal	RTN1	Saves the parameters to a SUBR by PUSHing a stack element.
POP	internal		POP's an element off the stack.
RECOVER	internal		LEN holds the error code. Prints the code, the function being executed, the arguments, and pauses to the operating system.
NCOVER	internal		LEN holds the error code. Prints the error co and pauses to the operating system.

1. Internal: used by SUBR's; SUBR: LISP SUBR; interp.: basic interactive cycle and/or initialization.

2. Standard LISP 7/16 linkage conventions.