COMUNICACIONES TECNICAS

Vol. V

1974 Serie B: Investigación No. 70

IMPLEMENTATION OF LISP 1.6 ON THE B-6700 COMPUTER.

Mario Magidin*
Raymundo Segovia*.

*Investigador del CIMAS.

Recibida 14-IV-74

Revisada por:
Robert Yates.

ABSTRACT

A description of the implementation of the LISP 1.6 language in the B-6700 computer of the National University of Mexico is presented, including a brief description of the interactive facilities and the functions implemented.

INDEX

1. Introduction.

2. An Overview of the System.

3. Description of the Interpreter.

4. Description of System Functions and Atoms.

5. Interactive Facilities.

6. Use and Generation of the System.

7. Acknowledgments.

8. Bibliography.

INTRODUCTION

The LISP (LISt Processing) Language was developed in 1958 by a group headed by Prof. J. MacCarthy at MIT [1]. Since then, LISP has become not only one of the most often used languages in symbolic manipulation, (specially in artificial intelligence, algebraic manipulation, etc.), but also, thanks to its elegant mathematical structure, a vehicle for most of the research in the area of automatic program synthesis and verification.

This paper is a description of the implementation of LISP 1.6 [2,3] on the B-6700 computer of the National University of Mexico. Although we have tried to be as clear as possible, some of it may have meaning only to those familiar with the language. For an introduction to LISP programming the reader is refered to [4] or [5].

For the sake of completeness, brief descriptions of the functions implemented and of the interactive facilities are included as well as instructions to use and generate the system. A more detailed description of the functions available can be found in [6]. The reader may also consult [9] for the results of a program testing all the features of the system.

AN OVERVIEW OF THE SYSTEM

LISP has been implemented in the B-6700 through an

Interpreter written in Burroughs Extended Algol [7]. The interpreter exists in two modes, batch and interactive or remote, the only difference being the editing and break packages available in the latter.

In either mode, the Interpreter operates in EVALQUOTE, reading a pair of s-expressions from its input file, and printing the evaluation on its output file, until an end of file condition is encountered on the input.

The system operates on an array of 64K words, divided into "spaces" of 256 words each. Spaces are allocated, dynamically for the different types of data the system handles, atoms, lists, large numbers and arrays.

Within each space, locations are assigned sequentially for only one type of data, and with the exception of arrays where segmentation is avoided, new spaces are assigned, only when the current one is used up.

An inventory of spaces is kept, as well as pointers to the next available cell of each data-type.

The system will recognize as an atom any sequence of characters, beginning with an alphabetic character, and not including any of the following ( ) , $ - + . [ ] or blanks.

The brackets are treated as "super-parentheses"; a right bracket supplies enough right parentheses to match back to the last left bracket or, if none is present, to match the first left parenthesis of the expression. It is important to point out, that this system does not have dotted pairs.

Special atoms, i.e. those including ilegal characters may be read with the $dsd syntax, where d is any character not is s, the print name of the special atom.

Numbers can be written either in decimal or in octal (appending a Q), and may be signed or unsigned, integers or reals; the exponential notation is valid.

There are two kinds of numbers in the system, the so-called "small numbers" and "large numbers".

Small numbers are integers whose absolute value is less than 2^{20}. They are stored as pointers outside the available space, i.e. positive small number n is represented as $2^{20} + n$, and negative small number n as $2^{21} + 2^{20} + |n|$.

Large numbers, i.e. any non-small number, are stored in the normal B-6700 format, one per word; no attempt to avoid duplication is made.

Unless expressely indicated not to, the system will always attempt to store numerical results as small numbers.

The system allows the user to declare arrays of any syze and dimensions. The elements of the arrays may be of any type, and furthermore, they may be mixed. Multidimensional arrays are implemented as arrays of arrays.

Arrays of less than 510 words are never segmented.

On the remote version, the system offers two interactive features, the editing and the break pa ages.

The editing package, allows the user to modify the definition of functions and property lists, while the break

package provides for selective insertion of interruptions or "breaks" at any point in the definition of a function. At this break, the user may inquire the value of any atom, assign new values, or evaluate any function, having full command of EVAL, before continuing the original evaluation.

The typing of "control - A", unwinds the system to the last ERRORSET, or to the top level, if none is present. The same effect is gotten by typing the "BREAK" key during output.

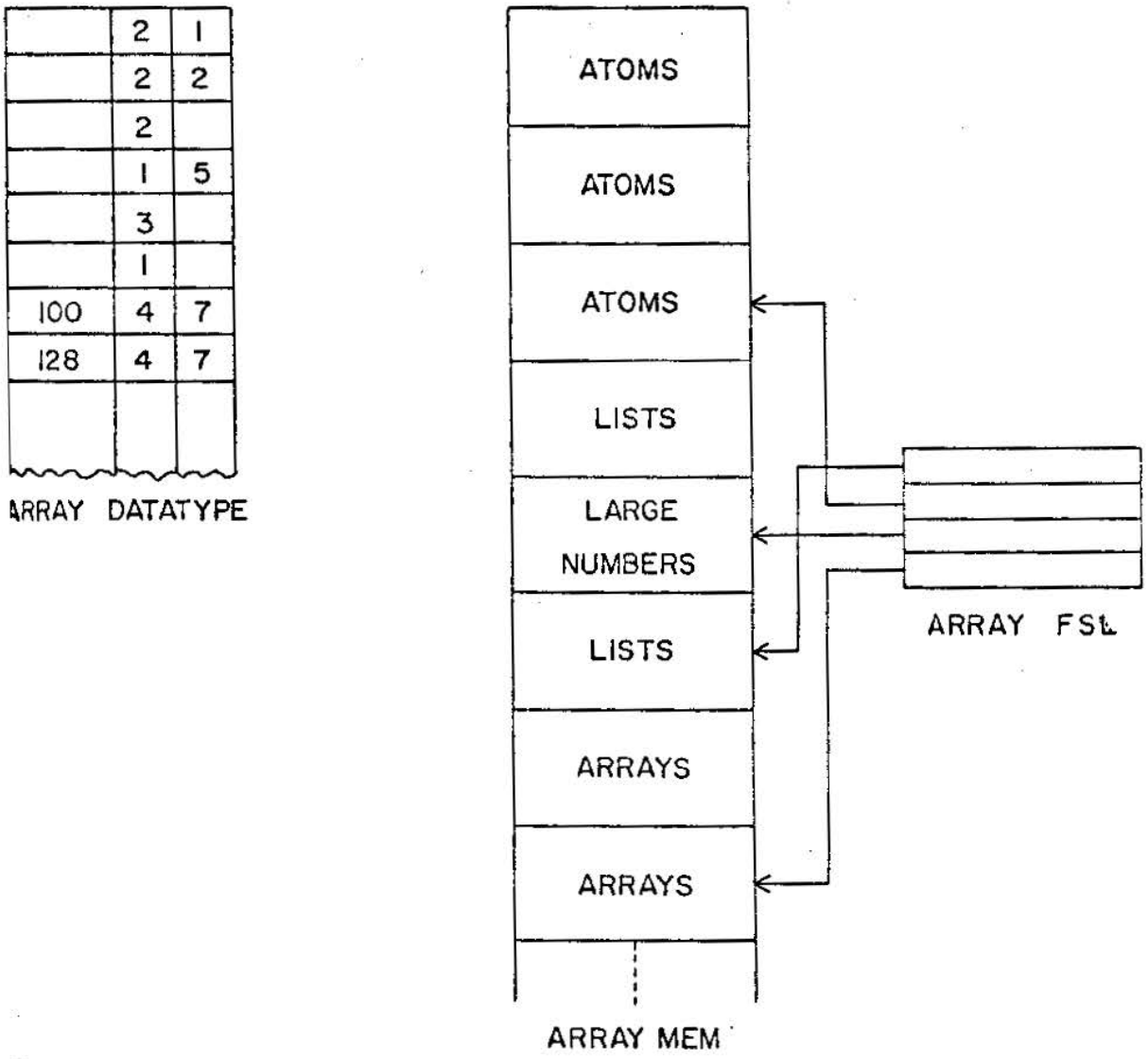
DESCRIPTION OF THE INTERPRETER

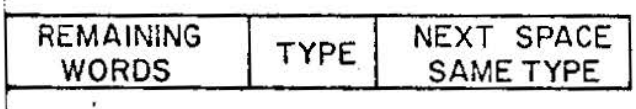
SYSTEM ARCHITECTURE.

The Interpreter makes use of the following data structures as its own tables and pointers.

MEM — This array of 64 K words holds all the LISP structures. All the LISP pointers are indices within this array. As mentioned before MEM is divided into "spaces" of 256 words each, for the different LISP data-types. It should be noted that, as words in MEM are assigned sequentially within a space, themselves assigned sequentially also, there is no need for a free storage list, only a set of pointers to the next available word of storage, per datatype; on the other hand, this requires a compacting garbage collector. (The size of MEM may be modified, changing the defined identifier MEMORIA).

DATATYPE — Datatype is a one-dimensional array of 256 words, one per space in MEM, identifying the kind of data in the corresponding space. To help the compacting phase of the garbage collector, pointers linking spaces of same type are kept. Finally, the number of words unused in a space of arrays is also saved. The format of the words in DATATYPE is shown in fig. 1.





WORD OF DATATYPE

FIG. 1 MEMORY ORGANIZATION

INDEX The variable INDEX points to the next available word in DATATYPE, i.e. to the next available space.

FSL This 4-word array holds a pointer to the last used word of MEM of each type, or a zero if none has been allocated so far.

OBLIST Atoms with same hashing are linked together. The hashing heads are held in OBLIST, a one-dimensional array of 128 words. New atoms are allways inserted at the head of the list.

BUFF This is a two-dimensional array that serves as I/O buffer, one array row per file. Their size is modified when a new file is created.

NUMCOL The number of characters so far written or read from buffer BUFF[I,*] are kept in NUMCOL[I].

PBUFFIN, PBUFFOUT These are pointers (in ALGOL sense), pointing into the current input and output buffers respectively. PBUFFIN points to the next character to be read, while PBUFFOUT points just ahead of the last character written.

LINEA, PLINEA. As the size of an I/O buffer is variable , PLINEA [I] holds the number of words of buffer BUFF [I]. For convenience and speed, the number of characters are held in LINEA [I].

ARGS The array ARGS is used as a stack, where old bindings of arguments in lambda expressions, as well as locals and labels in prog's are saved.

Each word of ARGS holds two pointers, one to the old binding, and one to the atom, to ease re-binding when exiting a lambda or prog expression. At each function call, a marker is inserted in ARGS acting as a separator. This marker holds a pointer to the name of the function called, a link to the previous marker, and a distinguishing mark. (See figs. 9 and 10).

B — The index in ARGS of the last marker is saved in B.

S — Variable S points to the top of the stack ARGS.

EJEC — The expression currently under evaluation is pointed at by EJEC.

STACKE — In the interactive version, the editing routines make use of array STACKE as a stack, when traversing the list structure being edited. To facilitate the work of the editor, it holds both a pointer to the head of the sublist, and a number, the index of this sublist in the list minus one. (See fig. 2).

PE — Pointer to the top of STACKE.

MACRODEF — Contains the text associated with the macro-instructions defined in the editing routine.

LASTDEF — Points to the next available byte in MACRODEF.

(LAMBDA (X) (IF (ZEROP X) 1 .(TIMES X (F(DECR X)))))

FIG. 2a. ORIGINAL EXPRESSION

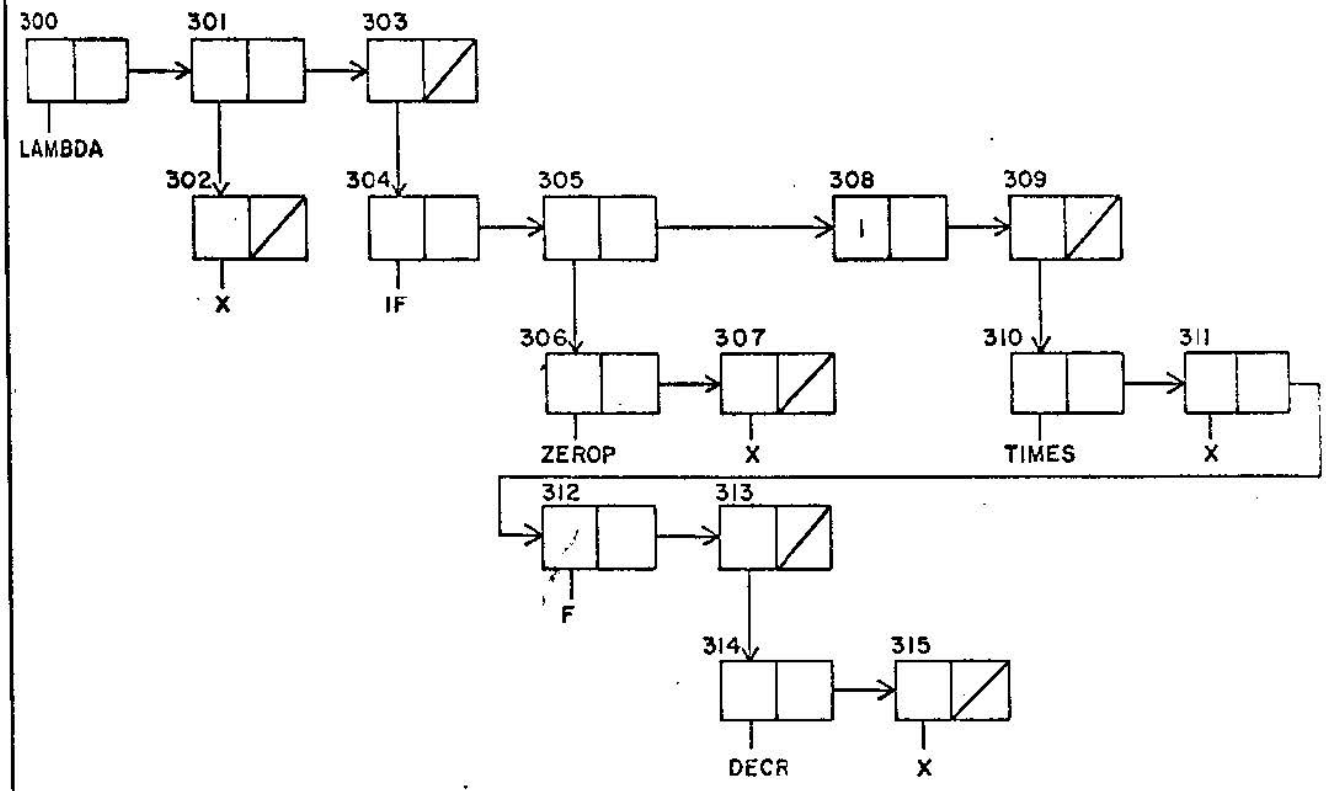


FIG. 2b. ORIGINAL EXPRESSION IN MEM

2	310
3	304
2	300

FIG. 2c. STACKE, WHEN THE EXPRESSION UNDER ANALYSIS IS (F (DECR X))

DATA TYPES.

The format of the datatypes handled by the system in MEM is as follows:

List nodes use one word of storage to house the 2 pointers, "CAR" in field [45:22] and "CDR" in [21:22] plus a space for the garbage collection mark in [47:1]. (See fig. 3).

An atom uses at least 3 words of storage, according to the format illustrated in fig. 4, where:

The "fun" field ([28:9]) carries a number that uniquely identifies system functions. Program defined functions carry their definition as first element in the property list. "trace" ([46:1]) is on when a trace of this atom is desired; in those cases "# recur" ([37:9]) will hold the recursion level for that function. "funarg" ([23:1]) indicates a functional argument (see evaluation). Finally, "oblist" ([19:20]) is used to link atoms with same hashing.

The organization of arrays is indicated in fig. 5 and fig. 6, for unsegmented and segmented arrays respectively. In both cases, each segment carries a header word with: the declared lower bound of the array in field "liminf" ([45:13]), segment size in field "dim" ([32:9]) and a special mark (bit 23) which is on for headers. Segments of the same array are linked, in a circular list, via the field link ([21:22]), the last segment having the segmentation bit (bit 22) off. Unsegmented arrays point to themselves and the segmentation bit is off.

FIG. 3 LIST NODE

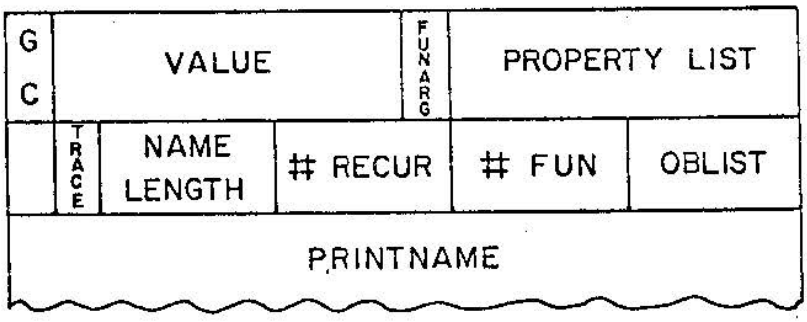


FIG. 4 ATOMS

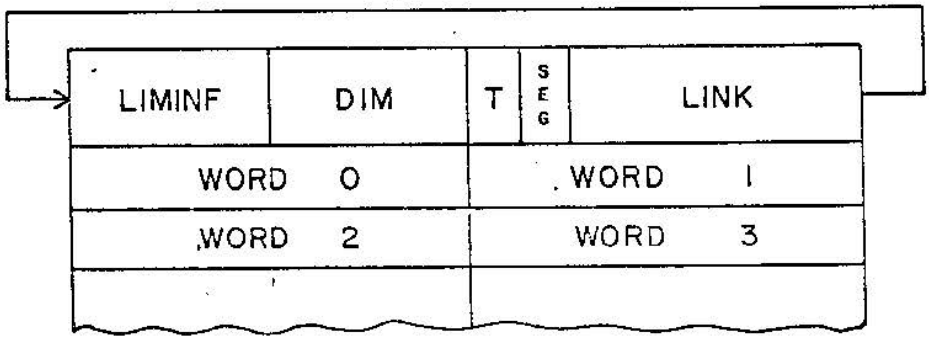


FIG. 5 UNSEGMENTED ARRAY

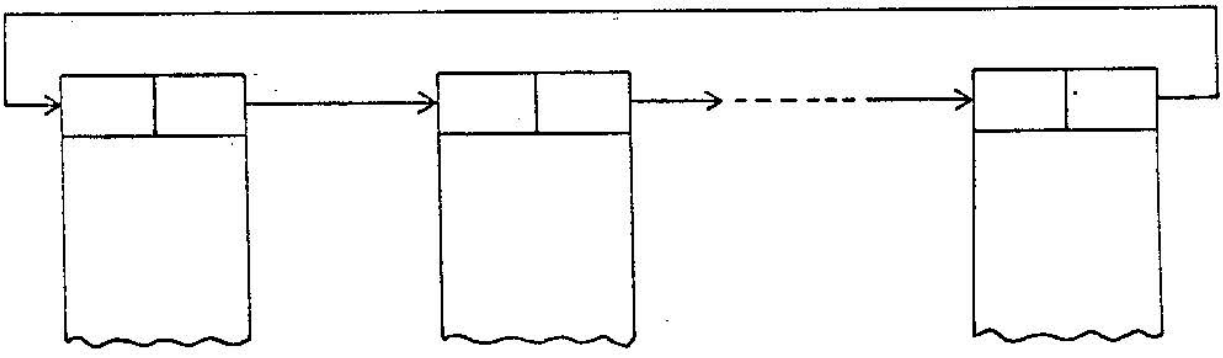


FIG. 6 SEGMENTED ARRAY

PROCEDURES.

Initialization and Miscellaneous Routines.

Initialization of the system is carried out by procedure INICIALIZA. This procedure fills in the system predefined atoms and functions into array MEM, setting up in an appropriate manner the hashing heads in OBLIST, as well as DATA TYPE, FSL and INDEX. Finally INICIALIZA sets up the I/O buffers.

DUMP. Procedure DUMP is used to dump array MEM. Its argument is the index to a file.

ERROR. ERROR is used to print out error messages.

Storage Handling Routines.

Free storage cells are provided to the system by procedures LIFT (for nodes and numbers), LIFTATOM (for atoms) and LIFTARRAY (for arrays). The operation of the first two is straightforward. LIFTARRAY will never segment an array of less than 510 words. It will first search in any previous array spaces for available storage, and if none is found it will ask for a new space to be allocated.

New spaces are allocated by procedure GETSPACE, which keeps DATATYPE up to date. When no more free storage is available, GETSPACE calls on GARBAGE to perform the garbage collection and compactification.

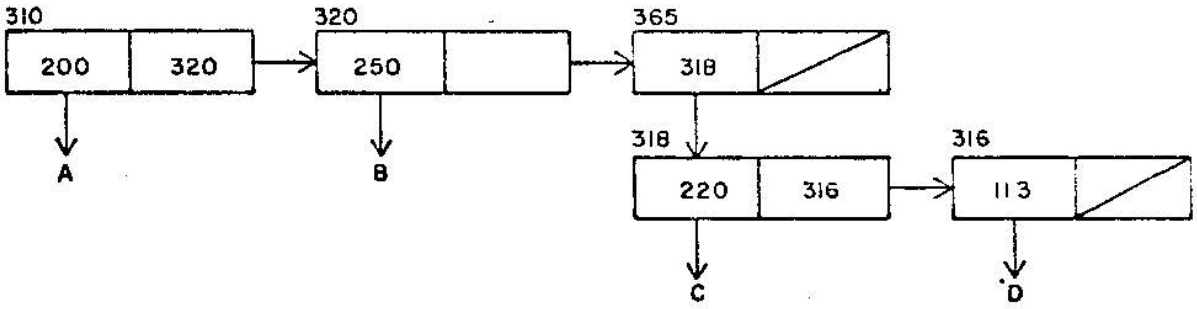
The compacting garbage collection is done in four steps: the first is the marking phase, where all the elements taking part in the computation are marked to separate them from the garbage. The structures to be saved are pointed at by EJEC, the elements from the stacks used for evaluation (argument stack, Algol stack) and the atoms (including their values and property lists). This is done by the recursive procedure MARCA which traverses a list structure marking the elements and Espol written intrinsics that scan the stack for list pointers.

The second is the moving stage, where all marked elements are copied sequentially, per data type, into a duplicate of array MEM (MEM1), modifying the elements in MEM, to point to the place where they now lie in the duplicate. A duplicate of DATATYPE is also constructed in this phase.

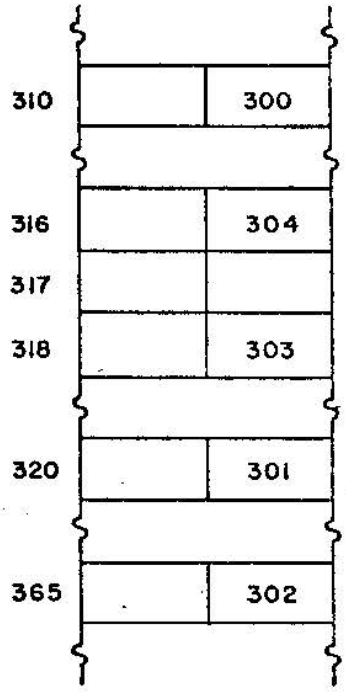
The third phase, updates all the pointers in MEM1, making use of the addresses left in MEM. Figure 7 illustrates this phase. Once more, Espol written intrinsics are used to modify the contents of the machine's stack.

Finally, the contents of MEM1 is copied back into MEM.

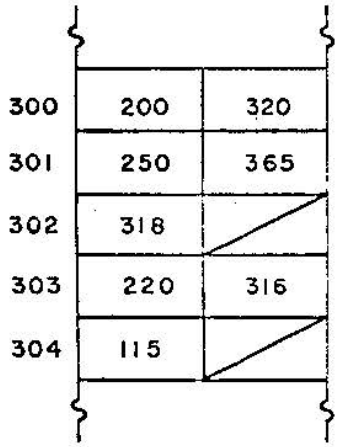
In a future version of the system, this will be handled by exchanging the appropriate data descriptors, and letting the MCP take care of removing the now unwanted array MEM1 (local to procedure GARBAGE).



a) INITIAL STATE (A,B,C AND D ATOMS)

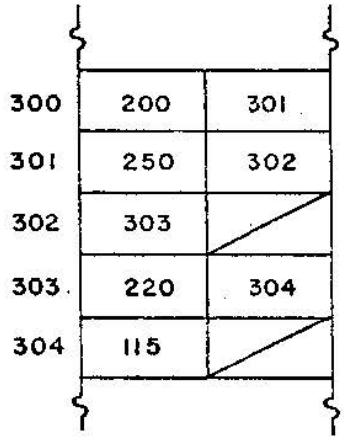


MEM



MEM I

b) MEM AND MEM I AFTER THE MOVING INFORMATION PHASE



c) MEM I AFTER UPDATING POINTERS (WE ASSUME THAT THE ATOMS IN 200, 220 AND 250 WERE NOT MOVED)

FIG. 7 OPERATION OF THE GARBAGE COLLECTOR

Input Routines.

The input routines are driven by procedure LEE. Its arguments are a pointer where to start constructing the list structure, a boolean variable indicating whether one or two s-expressions are to be read and the input file.

LEE will in turn call on TERMINALISTA (for a "]"), CREALISTA (for a "("), CREANUMERO (for a number), FINDELISTA (for a ")"), CREATOMO (for an atom) and CREASPECIAL (for a "$").

The last two procedures will call on INSERTATOMO to either create the atom if not present in the system, or get a pointer to it.

It is important to note that while reading an s-expression, the list structure is being constructed like a threaded list, with the last element always pointing to the previous level (see fig. 8).

These threads are removed when the expression is completed.

Also among the input routines is procedure LEECH, to read a character at a time.

Output Procedures.

The output procedures are driven by IMPR and IMPR2. The only difference between them, is that IMPR2 will output

(A B(C D E)(F G(H I)))

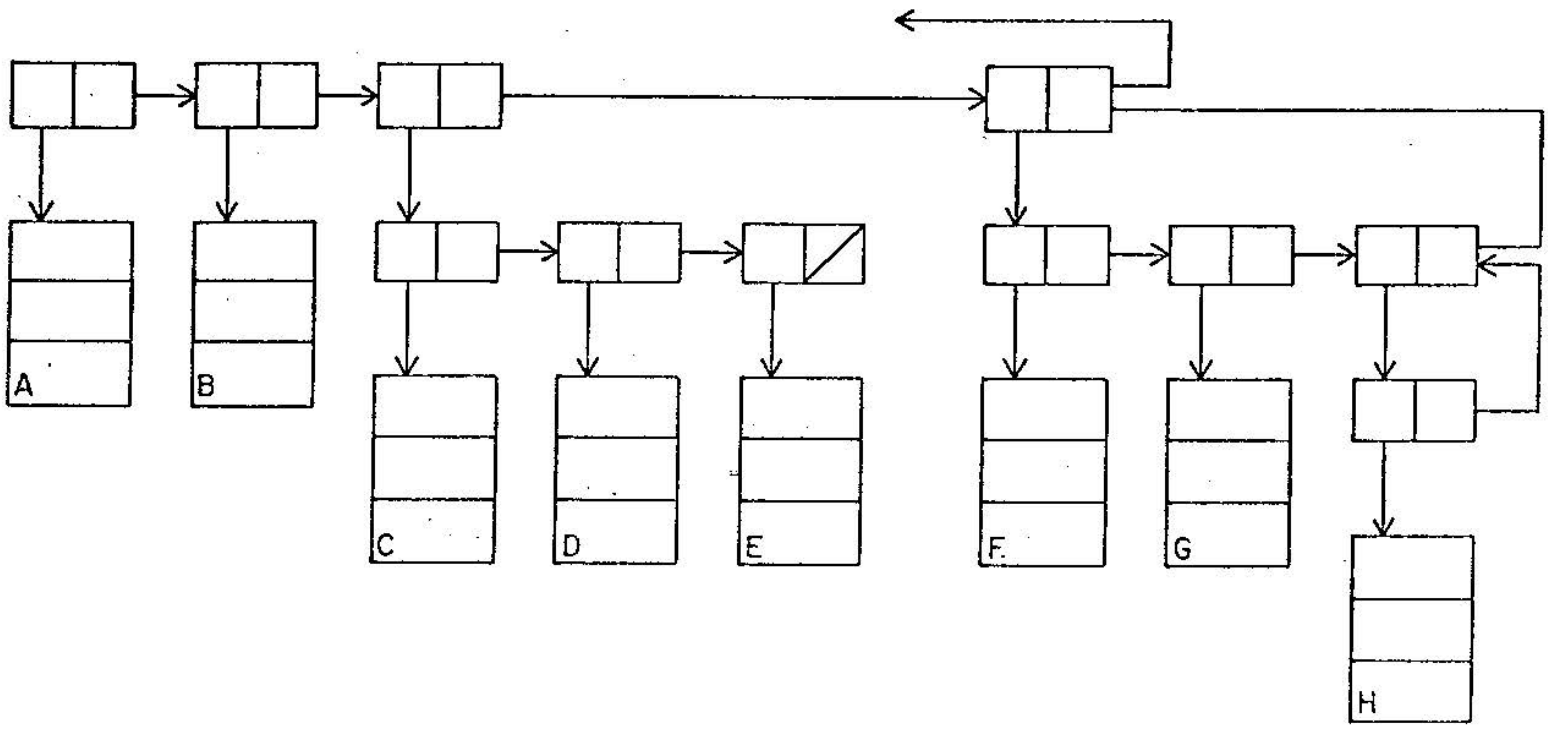


FIG. 8 LIST STRUCTURE AS IT LOOKS AFTER THE ATOM H HAS BEEN READ.

special atoms in a format acceptable by the input routines.

IMPR (IMPR2) calls on PRINT (PRINT2) to recursively print a list structure. In turn PRINT (PRINT2) calls on PRINTNUM1 for small numbers, PRINTNUM2 for large numbers, PRINTATOM (PRINTATOM2) for atoms and PRINTARRAY for arrays.

Also part of the output procedures are ESCRIBE to print a line, IMPCH to print a character, ESYZE and PRETTYPRINT to do the prettyprint and IMPTRACE to printout the tracing of atoms and functions.

Evaluation.

Evaluation is performed basically by EVAL, which recursively evaluates list expressions. EVAL is assisted by procedures EQUAL to evaluate (EQUAL X Y), GETD to obtain the address of a given system function, EVALQUOTE, which converts list structures read in evalquote notation to eval notation and by ARREGLOS, to create the declared arrays.

The system uses the so-called "shallow approach" [8], to store the values of atoms, (i.e., only the last binding is stored at the value cell of an atom).

When a lambda expression is encountered, the arguments are evaluated, previous bindings of the parameters are saved in ARGS and new binding are made. On exiting, the parameters are rebound to their previous values. A similar

approach is used in prog expressions for locals and labels.

Functional arguments are detected by the Interpreter, and the bindings are done in the property list cell instead of the value cell.

The functional argument problem is solved providing FUNCTION with an optional second argument. This argument specifies certain variables whose values at that time are saved for future reference. The funarg bit of the function passed is set to 1 to alert the Interpret about this. (See figures 9, 10 and 11).

The current values of these variables are stored in the property list of the function, under a special header, and the "funarg" bit in set to one.

DESCRIPTION OF SYSTEM FUNCTIONS AND ATOMS

The B-6700 LISP allows the user to redeclare system functions, both to create a synonym and to assign a new meaning to a standard function. Also, standard atoms such as NIL or T may get new values assigned, but the user is advised against this, as the results are unpredictable.

The system is very flexible in handing arguments to functions, letting the user pass any number of them. Missing arguments are always assumed to be NIL (even in arithmetic functions), while the extra ones are evaluated in program defined functions and ignored in system functions.

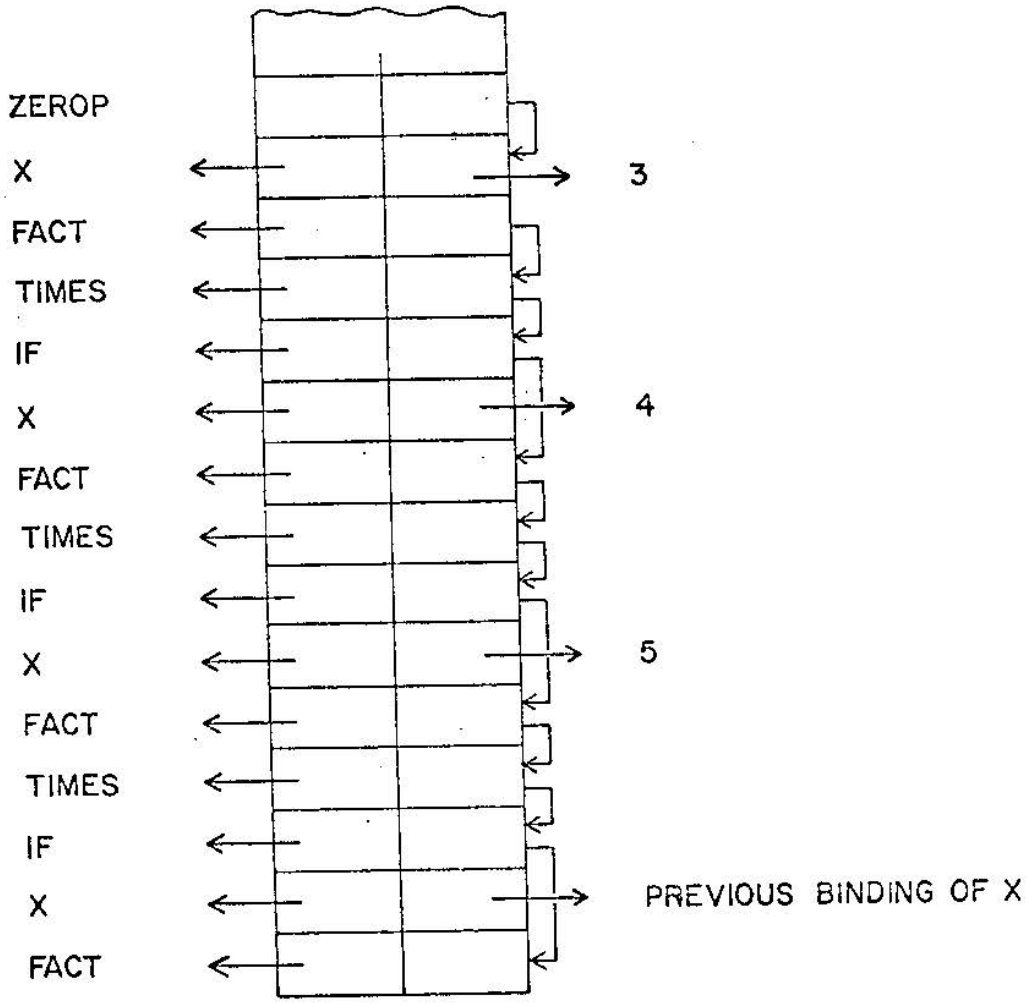


Fig. 9. State of ARGS during the evaluation of FACT(5) defined as (FACT (X) (IF (ZEROP X) 1 (TIMES (FACT (DECR X)) X))).

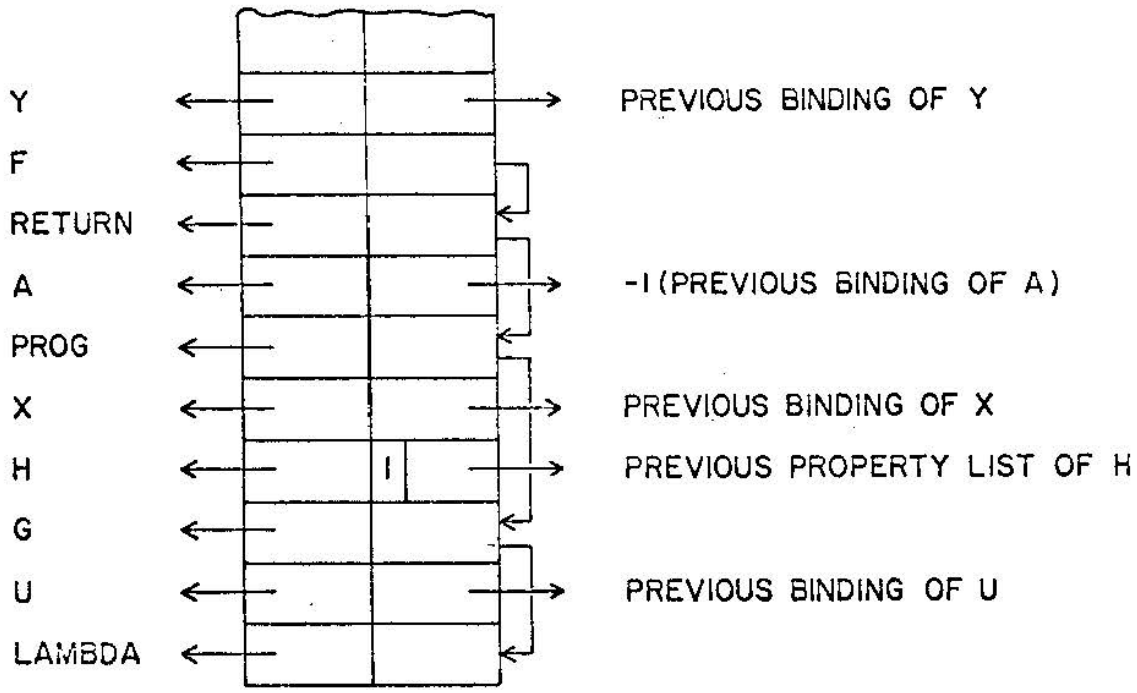


Fig. 10a. State of ARGS when entering F after the first call of G in the program of Fig. 11.

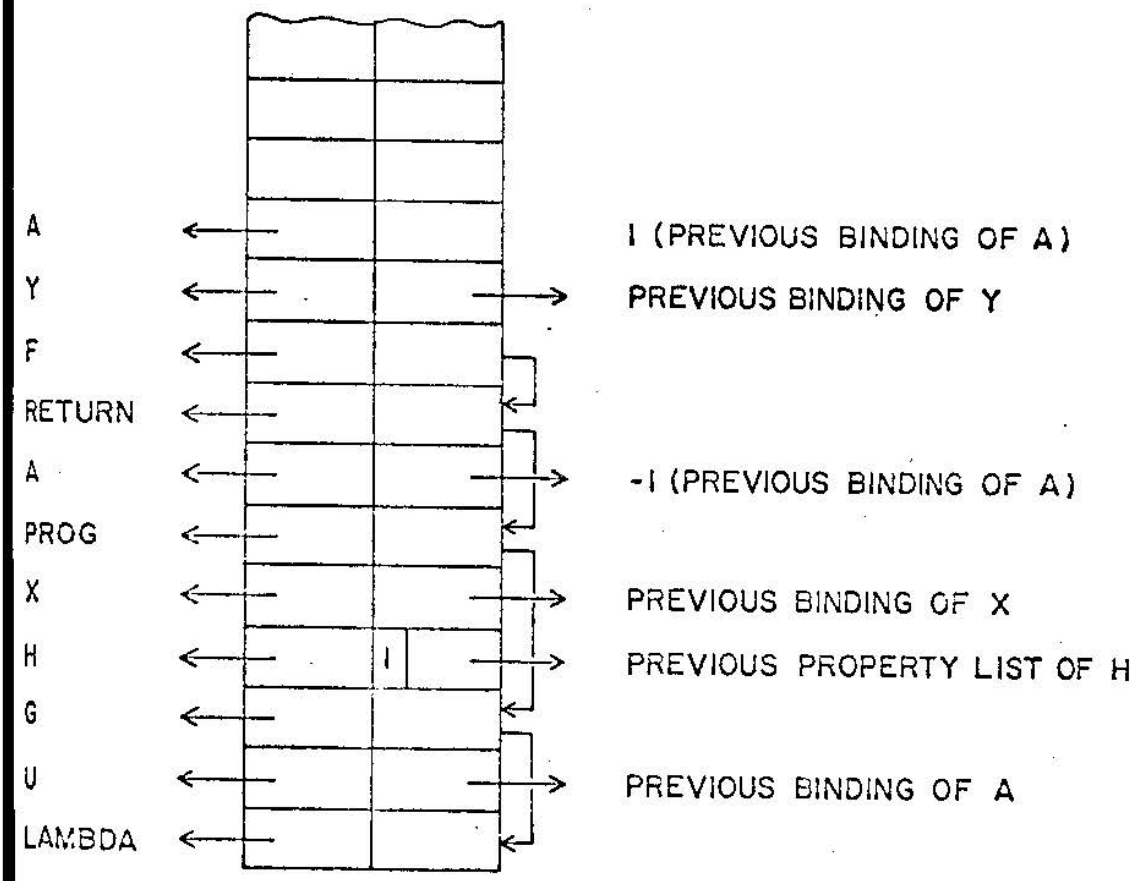


Fig. 10b. State of ARGS when entering F after the second call of G in the program of Fig. 11. Note that A has been rebound to -1 just before entering F, as a result of using (FUNCTION F (A)).

```
L I S P   B 6 7 0 0      VERSION 2.1(02/25/74)    03/20/74    12:32:09
*
47
DEFINE((
*
   (F(Y) (IF (MINUSP A) Y (MINUS Y))
*
   (G(H X) (PROG (A)
*
             (SETQ A 1)
*
             (RETURN (H X))
*
  ))
(F G)
*
SET(A -1)
-1
*
(LAMBDA (U) (G (FUNCTION E) U)) (3)
-3
*
(LAMBDA (U) (G (FUNCTION F (A)) U)) (3)
3
*
?END
#
```

Fig. 11. Simple LISP program showing the FUNARG problem. Note the use of the optional second argument of function in the second call of G and its effect.

SYSTEM FUNCTIONS.

Unless contrarily specified we shall use the letter A to denote non-numeric atoms and N to denote numbers (either small or large). In this case, we shall distinguish them by the use of an I if N is small and a B in case N is a large number. Letter L will usually stand for a list, M for an array and X will be used whenever the kind of object we are making use of is inmaterial for the function being considered (i.e., if the function is defined for any item already existing in the system). S-expressions will be denoted by an S.

To simplify even more our notation, our symbols so defined may appear one followed by another. By this we mean that the argument of the function may be of any of the kinds represented by the letters appearing in such a chain. For example, if we write "PRETTYPRINT (AL A)", we mean that the first argument of PRETTYPRINT may be either an atom or a list and that the second must be an atom; this, of course, would not exclude the case in which the second is omited: PRETTYPRINT (AL).

All conventions above remain valid for letters having indices.

Recall that any argument not present will be assumed to be NIL by the system.

To avoid repetition we shall only state explicity those functions in which no argument evaluation is made, and so, reference to arguments is really reference to the arguments' value. Also, if we do not state the contrary, we assume, the file "CARD" for input and "LINE" in output functions; these will also be usually the files used when the file designator of an input/output functions is omited or is NIL.

ABS(N).- Returns the absolute value of N.

ACCEPT(A).- Displays A on the "SPO" and causes the execution to be suspended until a response is keyed in at the SPO. Its value is that response.

ADD(A AI ALN).- Searches for the label AI in the property list of A. If found, ALN is added as an element of the property associated to AI (Thus, that property must be a list); otherwise ALN is put as a singlet at the end of the property list of A after the label AI.

AND(S_1 S_2...S_n).- Evaluates sequentially the S_i's. If any of them is NIL, execution terminates and the value of AND is NIL; otherwise the value is S_n. AND() is always T.

APPEND(L_1 L_2).- Makes a list with all the elements of L_1 followed by the elements of L_2.

ARRAY(LN_1 LN_2...LN_n S).- Returns an n-dimensional array created in the following way: if the i-th argument is a list, it must be of the form (N_{i_1} N_{i_2}) and then the i-th dimension of the array is created to have (N_{i_1} -N_{i_2} +1) elements, and having its lower bound equal to N_{i_1}. If it is a number, its efect is equivalent to having placed (0 N_i). All the elements of the array are assigned the value S, evaluated once for the whole array.

ARRAYP(X).- T if X is an array, NIL in any other case.

ARRAYSIZE(M).- Returns the number of elements of M; in case M is multidimensional returns its number of rows.

ARRAY*(LN_1 LN_2...LN_n S).- Same as ARRAY but S is evaluated again for each element of the array.

ASSOC(ALN L).- Searches in L for a sub-list having its CAR equal to ALN. If found, the value of ASSOC is such sublist, otherwise is NIL.

ATOM(X).- NIL if X is a list T otherwise.

BAKGAG(S).- Sets the error-print control equal to S, so that error messages are not printed while that control is T. Even though no error message is printed, unwinding always takes place. Returns previous setting.

BREAK(L).- L must be of the form (A_1 A_2...A_n), where each

of the A_i's has a definition as a LAMBDA or NLAMBDA expression; if such is the case, a call to BREAK1 is inserted in the definition of A_i after its argument list. This function is defined only in the remote version.

BREAK1(A L S).- If L and S are not present or if S is different from NIL, calls the break routine of the interpreter; otherwise nothing happens. Neither A nor L are evaluated. In the batch version, its effect is identical to COMMENT.

BREAKIN(A S_1 S_2).- Inserts a call to BREAK1 in the definition of A. S_1 is of the form (BEFORE L) or (AFTER L), where L is a pattern, and S_2 a condition; if S_1 matches with some part of the definition of A, (see editing command "B") the call to BREAK1 is inserted there and the value is T; in the other case, the value is NIL. Defined only in the remote version.

CAR(AL).- As usual its value is the first element of AL in case AL is a list. If AL is an atom, returns its value.

CDR(AL).- If AL is a list returns it with the CAR deleted, if it is an atom returns its property list.

C...R(AL).- All combinations of CAR and CDR up to the fourth level are included.

COMMENT (X).- Used to introduce comments in a program. Returns always NIL; X is not evaluated.

COND (L_1 L_2...L_n).- The L_i are of the form (S_{i_1},S_{i_2}...$S_{i_{k_i}}$ where the S_{i_1}, are predicates. The lists are considered sequentially until an S_{j_1}, is not NIL, then, S_{j_2} thru $S_{j_{k_j}}$ are evaluated and the value of the last one is returned; if k_j=1 the value of S_{j_1} is returned. If all the predicates are NIL, returns NIL.

CONS (S L).- Returns at list with CAR S and CDR L.

DECR (N).- Returns N-1.

DEFINE (L).- Each element of L is a list of at least two elements, the first always an atom. If there are only two elements the second is either a LAMBDA or NLAMBDA expression that becomes the definition of the atom, or is another atom, defining the first atom to be the same function as the second one. If there are more than two elements, they are the list of arguments and the body of a LAMBDA expression to be associated with the atom. Returns a list of the defined functions.

DEFLIST (L AI).- L is a list of pairs, atoms and properties. The properties are saved in the property list of the respective atom under the label AI.

DIFFERENCE (N_1 N_2).- Returns N_1 - N_2.

DISPLAY(A).- Displays A on the SPO, returns NIL.

DIVIDE(N_1 N_2).- Constructs a list with QUOTIENT(N_1 N_2) and REMAINDER(N_1 N_2).

DREMOVE(ALN L).- Searches in L for all the elements EQUAL to ALN. Returns L with those elements removed. (This function is destructive).

DREVERSE(ALN).- If L is a list, returns it with all its elements in reverse order; otherwise returns ALN. (This function is destructive).

DSUBST(ALN_1 ALN_2 L).- Searches in L for all the elements EQUAL to ALN_1. Returns L with those elements replaced by ALN_2. (This function is destructive).

DUMP(A).- Writes the main system array (MEM) on the file A, assumed to be a printer with MAXRECSIZE of at least 22 words.

EDIT(A).- Calls the system editor. Defined only in the remote version. A is the function to be edited.

ELT(M N_1 N_2...N_n).- Returns the N_1 , N_2 , ..., N_n -th element of M.

ENTIER(N).- Evaluates to the greatest integer less than or equal to N.

EQ(AI_1 AI_2).- T if its arguments are equal, NIL otherwise.

EQP(AN_1 AN_2).- T if its arguments are equal, NIL otherwise.

EQUAL(ALN_1 ALN_2).- T if its arguments are equal, NIL otherwise.

ERRORSET(S).- Evaluates S and returns a list with that value in case no error occurred during the evaluation; in other case returns NIL.

EVAL(S).- Evaluates S and returns that value.

EXPT(N_1 N_2).- Computes the N_2 th power of N_1.

FILE(A L).- Defines A as the file having the attributes indicated in L; these must appear in L as sub-lists on the form (attribute value). No mnemonics are allowed.

FIXP(X).- T if X is an integer, NIL otherwise.

FLOAT(N).- Converts N into a large number.

FLOATP(X).- T if X is a large number, NIL otherwise.

FMINUS(N).- FLOAT(MINUS(N)).

FPLUS(N_1 N_2...N_n).- FLOAT(PLUS(N_1 N_2...N_n)).

FQUOTIENT(N_1 N_2).- Returns N_1/N_2.

FTIMES(N_1 N_2...N_n).- FLOAT(TIMES(N_1 N_2...N_n)).

FUNCTION (AL L).- Used to pass functional arguments. It does not evaluate its arguments. AL is either the name of a function or a LAMBDA or NLAMBDA expression. If L is present, it is a list of atoms whose values at that point are saved, to be restored when the functional parameter to which AL is bound, is invoqued.

GENSYM( ).- Creates and returns an atom of the form GDDDDD, new at each call, where D is a digit.

GETD(A).- Returns the definition of A. If A is synonym of some function, returns that atom.

GETP(A AI).- Searches in the property list of A for the label AI; if found the value is the associated property, otherwise is NIL.

GO(A).- Transfers control to the expression following the label A (which is evaluated). Its use anywhere outside a PROG may cause unexpected results.

GREATERP(N_1 N_2).- T if N_1 is greater than N_2, NIL in any other case.

IF(S_1 S_2 S_3).- If S_1 is different from NIL returns S_2, otherwise returns S_3. Note that only one of them is evaluated.

INCR(N).- Computes N + 1.

(LAMBDA L S_1 S_2...S_n) L_1.- Defines a function whose independent variables are given in L. Also provides a one-to-one correspondence between them and the

elements of $L_1 = (S'_1\ S'_2 \ldots S'_m)$ which contains the arguments of the function, in the following way: If more formals (the elements of L) than arguments are found, the remaining are assigned value NIL; if, on the other hand, more arguments than formals appear, the remaining arguments are evaluated but ignored. Once the binding is finished, the S_i's are evaluated sequentially and the value is S_n.

LEFTSHIFT(N I).- Shifts I places to the left on the bits of the integral part of N. I must be less than or equal to 39.

LENGTH(ALN).- If ALN is a list returns its number of elements; or, ALN in the other case.

LESSP(N_1 N_2).- T or NIL according to whether N_1 is less than N_2 or not.

LIMINF(M).- Returns the lower bound of M.

LIST(X_1 $X_2 \ldots X_n$).- Makes a new list containing all its arguments.

LOAD(A S).- Reads an S-expression from file A. If S is different from NIL, that expression is considered as argument of DEFINE; if NIL, LOAD returns the read expression.

LOCK(A).- If A is a disk-file it is closed permanently; if it is a tape-file LOCK rewinds it.

LOGAND(N_1 $N_2 \ldots N_n$).- Evaluates the logical operation "and" of its arguments.

LOGOR(N_1 N_2...N_n).- Evaluates the logical operation"or" of its arguments.

LOGXOR(N_1 N_2...N_n).- Evaluates the logical operation "exclusive or"of the N_i's.

MAP(L S).- S must be a functional argument; applies S to L and then to the successive CDR's of L until L is NIL.

MAPC(L S).- S must be a functional argument; applies S to all the elements of L. Returns NIL.

MAPCAR(L S).- Similar to MAPC, but a list is constructed with the resulting evaluations.

MAPLIST(L S).- Similar to MAP, but makes a list with the resulting evaluations.

MAX(N_1 N_2...N_n).- Returns the greatest of the N_i's.

MEMB(AI L).- Searches in L for an element EQ to AI; if found, the value of MEMB is the CDR segment of L whose CAR is AI. In the other case is NIL.

MEMBER(ALN L).- Same as MEMB, but searches with EQUAL.

MIN(N_1 N_2...N_n).- Returns the smallest of the N_i's.

MINUS(N).- Evaluates -N.

MINUSP(N).- T or NIL according to whether N is less than zero or not.

NCHARS(A).- Computes the number of characters of the name of A.

NCONC(L_1 L_2).- Includes the elements of L_2 after the last element of L_1 in L_1. (This function is destructive).

NEQ(AI_1 AI_2).- NOT(EQ(AI_1 AI_2)).

(NLAMBDA L S_1 S_2'...S_n)(S_1' S_2'...S_m').- Same as LAMBDA except that the S_i''s are not evaluated.

NOT(S).- EQ(S NIL).

NTH(L N).- Returns the N-th element of L.

NULL(S).- Identical to NOT.

NUMBERP(X).- T or NIL according whether X is number or not.

OR(S_1 S_2...S_n).- If S_i is different from NIL, execution terminantes and the value of OR is S_i; otherwise S_{i+1} is considered. OR( ) Is always NIL.

PACK(L).- L is of the form (AN_1 AN_2...AN_n). Returns an atom whose printname is the concatenation of the AN_i's.

PAGE(A).- Performs a skip to the top of the next page of file A.

PLUS(N_1 N_2...N_n).- Returns N_1 +... +N_n. If n = 0 returns 0.

PRETTYPRINT(AL A).- If AL is an atom its definition is prettyprinted on file A; otherwise it proceeds as before with each element of AL.

PRINT(X).- Prints X on file LINE changing line.

PRINTCH(A_1 A_2).- Prints the first character of A_1 on file A_2 following the previous contents of that line.

PRINTLEVEL(I A).- Sets the print level of file A to I. Returns previous setting.

PRINT1(X A).- Prints X on file A without changing the record.

PRINT2(X).- Identical to PRINT, only that special atoms are written in a way acceptable to the input routines.

PRINT3(X A).- Similar to PRINT1, but operating as PRINT2 on special atoms.

PROG(L S_1 S_2...S_n).- Initializes the elements of L to NIL, then evaluates sequentially the S_i's unless a GO is executed. Returns S_n unless a RETURN is executed. It is permissible to nest PROG and to transfer to an outer one from an inner one.

PROG1(S_1 S_2...S_n).- Evaluates the S_i's in sequence returning S_1.

PROGN(S_1 S_2...S_n).- Evaluates the S_i's in sequence returning S_n.

PROP(A ALN X).- Searches in the property list of A for an element EQUAL to ALN. If found returns the remaining of the property list, otherwise X.

PUT(A AI ALN).- Puts on the property list of A property ALN under label AI.

QUOTE(X).- Returns X unevaluated.

QUOTIENT(N_1 N_2).- Computes the integral part of N_1/N_2.

READ( ).- Reads an S-expression from file CARD.

READATT(A_1 A_2).- Returns the value of attribute A_2 in file A_1.

READCH(A).- Reads a character from file A.

RECLAIM( ).- Causes a garbage collection.

REMAINDER(N_1 N_2).- Computes N_1 mod N_2.

REMOVE(ALN L).- Creates a list with every ocurrence of ALN removed from L.

REMPROP(A AI).- Removes label AI and the property associated to it from the property list of A.

REPLACA(AL ALN).- Replaces the CAR of AL with ALN.

REPLACD(AL L).- Replaces the CDR of AL with L.

RETURN(X).- Exits the PROG with value X.

REVERSE(ALN).- Creates a list whose elements are the elements of ALN in inverse order.

REWIND(A).- Rewinds file A.

RIGHTSHIFT(N I).- Shifts I places to the right the bits of N. I has to be less than 39.

SELECTQ(AI L_1 L_2...L_n X).- Each L_i is of the form (AL_i Si_1 Si_2...Sin_i) and they are not evaluated. If AL_i is an atom and equal to AI the Si_js are evaluated and the value of SELECTQ is Sin_i. If AL_i is a list and AI is MEMB of AL_i the same procedure is applied, otherwise L_{i+1} is taken. If none is satisfied the value of SELECTQ is X.

SET(A X).- The value of A become X. If X is of the form (FUNCTION AL), the property list of A is replaced by that of AL.

SETA(M N_1 N_2...N_n X).- Assigns to the N_1 , N_2,...,N_n -th element of M the value of X. The dimension of M is at least n.

SETATT(A L).- Adds to A the attributes and values indicated in L (see FILE).

SETQ(A X).- SET(QUOTE(A)X).

SETQQ(A X).- SET(QUOTE(A) QUOTE(X)).

SPACE(A I).- Advances I records in the file A.

SQRT(N).- Computes the square root of ABS(N).

SUBST(ALN_1 ALN_2 L).- Creates a list with every appearance of ALN_1 in L replaced by ALN_2.

SYSFILES( ).- Returns a list of the form (A_1 A_2...A_n), where each of the files LISPSYSFILE/A_i is present in the user's library. (See SYSIN, SYSOUT).

SYSIN(A).- Reads from the sysfile named LISPSYSFILE/A. (See SYSOUT).

SYSOUT(A).- Performs a dump of the interpreter's memory into sysfile LISPSYSFILE/A. This allows the user future recovery at this point.

TEREAD(A).- Terminates inputing from the current record of file A.

TERPRI(A).- Terminates outputing into file A. Repeated use of TERPRI outputs blank lines.

TIME(I).- Same as the time function of the system.

TIMES(N_1 N_2...N_n).- Computes the product of the N_i's. If n = 0, returns 1.

TRACE(L).- L is of the form (A_1 A_2...A_n). Sets up a trace of functions (or atoms' value) A_i. Every change in value, call to or return from A_i will be printed.

UNBREAK(L).- L is of the form $(A_1\ A_2 \ldots A_n)$. Removes the call to BREAK1 from the definition of A_i as put by BREAK.

UNBREAKIN(A L).- Removes the call to BREAK1 in A set by BREAKIN at L. (See BREAKIN for the form of L).

UNPACK(A).- Creates a list whose elements are the letters of the print-name of A.

UNTRACE(L).- L is of the form $(A_1\ A_2 \ldots A_n)$. Removes the tracing mark set by TRACE.

WRITE(X A S).- If S is NIL it will output X on file A; otherwise a list whose elements are lists containing both the name and the definition of each element of X is written on A (as LOAD requires).

ZEROP(N).- T or NIL according to whether N is zero or not.

SYSTEM ATOMS. The predefined atoms in the system are:

Name	Value
ARROW	↑
BLANK	
COMMA	,
DASH	-
DOLLAR	$
DOT	.
LBRACK	[
LPAR	(
NIL	NIL
PLUSS	+
RBRACK	]
RPAR	)
SLASH	/
STAR	*
T	T

The following table, lists the file attributes implemented in the system, and the type of object the attribute accepts or returns.

Name	Type
ATTERR	Predicate
ATTVALUE	Number
ATTYPE	Number
BLOCK	Number

BUFFERS	Number
DATE	Number
DENSITY	Number
EOF	Predicate
EXTMODE	Number
FILETYPE	Number
FORMMESSAGE	Atom
KIND	Number
MAXRECSIZE	Number
MYUSE	Number
OPEN	Predicate
PARITY	Number
PRESENT	Predicate
PROTECTION	Number
RECORD	Number
RESIDENT	Number
SAVEFACTOR	Number
SECURITYTYPE	Number
SECURITYUSE	Number
TITLE	Atom

INTERACTIVE FACILITIES

THE EDITOR. As mentioned before, a call to the function EDIT causes the interpreter's editor routine to be activated. The argument of EDIT is an atom whose definition is to be edited. The editor answers with the ready character ":"; whenever this character appears, any number of commands can be sent provided they do not exceed a line. These commands are sequentially executed unless an error occurs, in which case the erroneous part will be written followed by a "?", and the remaining ignored.

A very useful feature of the editor is the macro facility that allows the user to define a parametrized collection of editing commands. This definition may then be simply invoqued or applied repeatedly to an expression until an error occurs.

Most of the editor's commands refer to a part of the expression being edited. That part is called "the current expression" and will be denoted by EP. Initially EP is set to the definition of the argument of EDIT. We now describe the commands defined: (we still follow the notation introduced before unless quoted).

It should be pointed out, that commands implying a modification of the list structure are destructive, i.e., actual REPLACA, REPLACD and NCONC are performed.

I.- Sets EP to that element of EP indicated by I. In

this and in any case in which EP is tried to be set out of bounds, an error will occur but EP will not be modified.

"-".- Moves EP one level upwards.

"/".- Moves EP to the level at which edition began.

"<".- Moves EP one site backward at the same level.

">".- Moves EP one site forward at the same level.

"=".- Sets EP to the fragment of the list containing EP and whose CAR is EP. Example:

```
: P 3 = P
( A B C D E )
... ( C D E )
```

The printing routine identifies a fragment printing ... before it.

"#" A L $cSTRINGc.- Defines a macro-instruction A by associating the string between the delimiter "c" with A. L must be either vacuous (i.e., a blank or ()) or a list of formal symbols separated by commas. The formal symbols must be atoms ($-notation is allowed). Delimiter "c" may be any character not contained in the string. These definitions are saved in the property list of A and may be reused without definition in another call to the editing routine.

"!" A L.- Invocates the text associated with A. L is a list of the form (X_1,X_2, ... ,X_n); where each X_i is either any string not containing ")" and commas or else a string of the form $cSTRINGc; where, as before, c is any character not contained in the string.

"*" A L .- Similar to "! A L", but will iterate the macro-instruction (once the proper parameter replacement has been done) until an error occur during its evaluation.

Nested invocation.- Macro-instructions may be nested up to any (reasonable) level. If an error occurs during a nested invocation, the only text ignored will be that of the macro being executed (i.e., the one at the highest level) and execution will continue normally at the point following this last invocation.

Parameter conventions.- It is not strictly necessary that the number of parameters appearing in an invocation command coincides with the number of formals in the declaration. Parameters may be omitted provided that their site is indicated by two consecutive commas. Also, a right parentheses set after the i-th parameter, will cause the remaining n-(i+1) to be omitted. For example, the following syntax is correct:
! A (X_1,,,X_4, ... ,X_{n-m}). Whenever a parameter is omitted from the parameter-list, its site in the text

associated with the macro identifier will be filled with a blank space. Note that the parameters are not evaluated in any form, as the macros are just a text substitution method.

Use of dots.- The use of a dot in the definition of a macro-instruction will cause the parameters to be appended to the part of the text preceeding the corresponding formals whenever these appear in the form <any text>. X_i. For example, suppose we define A(X) as $$P.X$, then a call to A in the form A(L 3) will cause the replacement "PL 3" and calling A(P) will cause the replacement "PP". Note that defining A(X) as $$PX$ would not be equivalent, yielding "PX" in both cases. Examples:

1) Suppose we want to change all lists of the form (set (quote a) b) to (setq a b), then a procedure may be

```
HS$"B  (set (quote  ==)  ==)
    c1  setq
    ↑ 2
    Q 2"
```

2) To remove all the parentheses inside EP, we may proceed as follows:

```
# S1  $"*S2 1 = "
# S2  $" ↑ 1"
* S1
```

"Q" I.- Removes the I-th element of EP.

"I" I S.- Inserts S as de I-th element of EP.

"R" I S.- Replaces the I-th element of EP by S.

"A" I S.- Appends S to the I-th element of EP.

IN THE ABOVE FOUR COMMANDS THE MODIFICATIONS ARE DESTRUCTIVE

"E" S.- Evaluated the S-expression S.

"EQ" S_1 S_2.- Evaluates the EVALQUOTE-expression given by

the S_i's'

"P".- Prints EP.

"PL" I.- Prints EP changing provisionally the print level of LINE to I.

"PP".- Prettyprints EP.

"F".- Terminates edition.

"M" A.- Change EP to the definition of A.

"N" S.- Terminates current edition and starts a new one with S (after EVALuating it).

"⍐" .- Moves EP to the element immediately "following" it (in print order) no matter how many right parentheses need to be jumped; if none "follows" it, an error is produced (i.e., it goes up until EP can be advanced forward one place).

"(" I.- Inserts a left-parenthesis before the I-th element of EP, I = 0 is invalid; for example.

```
      :
P ( 2 P
( A B C D )
( A ( B C D ) )
      :
```

")" I_1 I_2.- Inserts a right-parenthesis after the I_2-th element of the I_1-th element of EP(I_1 = 0 is invalid); for example,

```
:
P ) 2 3 P
( A ( B C D E F ) G H )
( A ( B C D ) E F G H )
:
```

"[" I.- Removes the left parentheses before the I-th element of EP; e.g.,

```
:
P [ 1 P
( A B )
A
:
```

or

```
:
P [ 2 P
( A ( B ) C )
( A B )
:
```

"]" I.- Moves the right parentheses following the I-th element of EP to the end of EP; example,

```
:
P ] 2 P
( A ( B C ) D )
( A ( B C D ) )
```

"↑" I.- Removes both the left and the right parentheses from the I-th element of EP.

"%" I_1 I_2.- Inserts a left parenthesis before the I_1-th element and a right parenthesis after the I_2 element of EP. Of course, I_1 must be less than or equal to I_2.

ALL SIX COMMANDS ABOVE ARE DESTRUCTIVE

"B" AL.- Searches in EP for an expression matching with the pattern AL, if found, sets EP to that expression, in other case causes an error. (If AL is an atom, EP is set to the fragment beginning in that atom). AL may contain atoms, "&", "==" or more patterns. Any atom appearing in AL distinct from "&" and "==" will match with exactly the same atom contained in some expression, "&" will match with any s-expression and "==" will match with the CDR of any list.

"C" <option> AL S.- <option> must be either vacuous or "?", searches in EP all sub-expressions matching with AL. If the option was "?", the editor will write each of these, waiting for an answer and if that answer is different from a carriage return, nothing happens; otherwise the expression is replaced, if <option> was vacuous, the replacing is made unconditionally.

BREAK FACILITIES.

BREAK1(A L S).- Allows the programmer to stop execution at the moment BREAK1 is evaluated. If S is present the break routine of the interpreter is called provided that S is not NIL; otherwise, it is called always. In the first case, the break routine will write:

BREAK OF A < place > < pattern > WITH S,

where L = (< place > < pattern >), and<place>is either BEFORE or AFTER. The break routine is now ready to execute its own commands (the break ready character is "*"). The commands defined are:

"E" S.- Evaluates the S-expression.

"EQ" S_1 S_2.- Evaluates the EVALQUOTE-expression given by the S_i's.

"R" S.- Terminates the break returning value S.

"F".- R NIL.

USE AND GENERATION OF THE SYSTEM

The LISP system operates in two modes; batch and remote. The batch system is called by the following cards:

```
? RUN LISP/LISP
? DATA CARD
  (LISP program)
? END
```

For the remote versions type

```
E  * SYSTEM/LISP
```

and the system will answer writing a title and a ready character (*) indicating that it is waiting for input. Typing ?END terminates the session.

The sequence of cards to compile the source file -LISP/SOURCE- in order to generate the interpreter in the remote or batch mode are the following.

For batch:

```
? COMPILE LISP/LISP ALGOL LIBRARY
? ALGOL PROCESSTIME = 120; ALGOL IOTIME = 120,
? ALGOL FILE TAPE = LISP/SOURCE
? STACK = 2500
? DATA
$ SET MERGE RESET REMOTE
? END
```

and for the remote version

```
? COMPILE OBJECT/LISP WITH ALGOL TO LIBRARY
? ALGOL PROCESSTIME = 120; ALGOL IOTIME = 120
? ALGOL FILE TAPE = LISP/SOURCE
? STACK = 2500
? DATA
$ SET MERGE
$ SET REMOTE
? END
```

ACKNOWLEDGMENTS

The authors gratefully acknowledge the participation of Mr. Max Díaz in this project. He collaborated in many routines, and was particularly responsible for the implementation of macros in the editor and the prettyprint routines.

BIBLIOGRAPHY

1. J. Mc Carthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine. Part I. "Communications of the ACM", 3, pp 184 - 195, April 1960.

2. W. Teitelman, D. G. Bobrow, A.K. Hartley, D.L. Murphy. BBN-LISP, Tenex Reference Manual. Bolt, Beranek and Newman, Inc., July 1971.

3. Lynn H. Quam. Stanford LISP 1.6 Manual. Stanford Artificial Intelligence Project. SAILON 28.3, Sept. 1969.

4. C. Weissman. "LISP 1.5 Primer". Dickenson Publishing Company, Inc. Belmont Calif. 1967.

5. W.D. Maurer. "The Programmer's Introduction to LISP". American Elsevier Inc. New York, 1972.

6. M. Magidin, R. Segovia. Manual Preliminar del Sistema LISP B-6700. Comunicaciones Técnicas del CIMAS, 3, 16, 1972.

7. Burroughs B6700/B7700 Extended Algol Compiler. Form No. 5 000 136, June 1972.

8. J. Moses. The Function of FUNCTION in LISP. "SICSAM Bulletin", 15, p 13-17, July 1970.

9. M. Díaz. Las Funciones Definidas en el Sistema LISP B-6700. Comunicaciones Técnicas del CIMAS. (To be published).