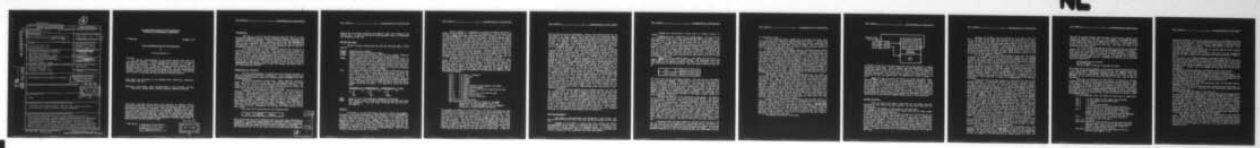
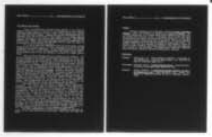
AD-A052 305 MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTE--ETC F/G 9/2
DATA REPRESENTATIONS IN PDP-10 MACLISP.(U)
SEP 77 G L STEELE

UNCLASSIFIED AI-M-420 N00014-75-C-0643 NL

| OF |
AD
A052305





END
DATE
FILMED
5 -78
DDC

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AI-M-420 (AIM 420)
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Data Representations in PDP-10 MacLISP.
5. TYPE OF REPORT & PERIOD COVERED: MEMOrandum rept.
6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Guy Lewis Steele, Jr
8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0643, E(11-1)-3070
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, Massachusetts 02139
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 14p.
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, 1400 Wilson Blvd, Arlington, Virginia 22209
12. REPORT DATE: Sep[illegible]977
13. NUMBER OF PAGES: 12
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Office of Naval Research, Information Systems, Arlington, Virginia 22217
15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT *(of this Report)*: Distribution of this document is unlimited.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES: None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*: data types, data representations, LISP, Mac LISP, garbage collection, storage spaces, storage allocation, dynamic allocation

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*:

The internal representations of the various MacLISP data types are presented and discussed. Certain implementation tradeoffs are considered. The ultimate decisionson these tradeoffs are discussed in the light of implementation of large systems such as MACSYMA. The basic strategy of garbage collection is outlined, with reference to the specific representations involved. Certain "clever tricks" are explained and justified. The "address space crunch" is explained and some alternative solutions explored.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED 407 483
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 420 September 1977

DATA REPRESENTATIONS IN PDP-10 MACLISP

by

Guy Lewis Steele Jr. *

Abstract:
The internal representations of the various MacLISP data types are presented and discussed. Certain implementation tradeoffs are considered. The ultimate decisions on these tradeoffs are discussed in the light of MacLISP's prime objective of being an efficient high-level language for the implementation of large systems such as MACSYMA. The basic strategy of garbage collection is outlined, with reference to the specific representations involved. Certain "clever tricks" are explained and justified. The "address space crunch" is explained and some alternative solutions explored.

This paper was presented at the MACSYMA Users Conference, Berkeley, California, July 1977.

Keywords: data types, data representations, LISP, MacLISP, garbage collection, storage spaces, storage allocation, dynamic allocation

This report describes research done at the Laboratory for Computer Science (formerly Project MAC) and at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. This work was supported, in part, by the United States Energy Research and Development Administration under Contract Number E(11-1)-3070 and by the National Aeronautics and Space Administration under Grant NSG 1323. Support for the Artificial Intelligence Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

* NSF Fellow

Introduction

MacLISP is a version of LISP which is used not only as a user application language but as a systems programming language, supporting such systems as MACSYMA and CONNIVER. As such, it has been carefully designed with speed as one of its major goals. Generality, ease of use, and debuggability have not been neglected, but speed of compiled code has been the primary consideration. This is a departure from the traditional view of LISP as a friendly and general but slow and clumsy language.

The representations of data objects in MacLISP have undergone a continuous evolution towards this goal. When MacLISP was first created, the data representations were designed for simplicity and compactness at the expense of speed. Since then there have been at least two major revisions, each to speed up compiled code and simplify the processing of the data. Here we discuss the current implementation on the PDP-10 (MacLISP also runs on Multics, and on the "LISP machines" being constructed at the MIT Artificial Intelligence Laboratory). We shall contrast it with previous MacLISP implementations and implementations of other LISP systems, and discuss some of the design decisions involved.

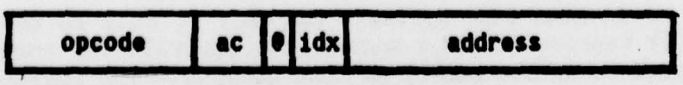
Organization of the PDP-10

The data representations in MacLISP have been carefully designed to take full advantage of the PDP-10 architecture. A full understanding of the design decisions involved requires the following minimal knowledge of the PDP-10 instruction set.

The PDP-10 operates on 36-bit words. Memory addresses designate words, not bytes, and are 18 bits wide; thus two addresses can fit in one word. There is a class of instructions which manipulate half-words; for example, one can store into half of a memory word and either not affect the other half or set the other half to all zeros or all ones.

The PDP-10 has 16 accumulators, each 36 bits wide. All but one can be used for indexing; all can be used as stack pointers; all can be used for arithmetic. The accumulators can also be referenced as the first 16 memory locations (though they are hardware registers and not actually memory locations). For reasons explained later, MacLISP devotes certain accumulators to specific purposes. Accumulator 0 contains the atom NIL. Accumulators 1-5 may contain pointers to data objects; these are used to pass arguments to LISP functions and return values from them. Accumulators 6-10 are scratch registers, and are generally used for arithmetic. Accumulator 11 is reserved for a future purpose. Accumulators 12-15 are used for stack pointers to the four stacks.

Every user PDP-10 instruction has the following format:



Each instruction has a 9-bit operation code and a 4-bit field specifying an accumulator. The effective memory address (or immediate operand) is uniformly computed by adding to the 18-bit address field the contents of the accumulator specified by the 4-bit index field (a zero index field means no indexing). If the indirection bit "@" is set, then a word is fetched using the computed

address and the process iterated on the address, index, and @ fields of the fetched word. In this way the PDP-10 allows multiple levels of indirection with indexing at each step.

MacLISP Data Types

MacLISP currently provides the user with the following types of data objects:

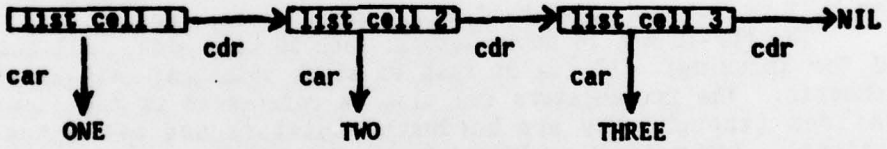
FIXNUM Single-precision integers.

FLONUM Single-precision floating-point numbers.

BIGNUM Integers of arbitrary precision. The size of an integer arithmetic result is limited only by the amount of storage available.

SYMBOL Atomic symbols, which are used in LISP as identifiers but which are also manipulable data objects. Symbols have value cells, which can contain LISP objects, and property lists, which are lists used to store information which can be accessed quickly given the atom. Symbols are written as strings of letters, digits, and other non-special characters. The special symbol NIL is used to terminate lists and to denote the logical value FALSE.

LIST The traditional CONS cell, which has a CAR and a CDR which are each LISP objects. A chain of such cells strung together by their CDR fields is called a list; the CAR fields contain the elements of the list. The special symbol NIL is in the CDR of the last cell. A chain of list cells is written by writing the CAR elements, enclosed in parentheses. A non-NIL non-list CDR field is written preceded by a dot. An example of a list is (ONE TWO THREE), which has three elements which are all symbols. It is made up of three list cells thus:



ARRAY Arrays of one to five dimensions, dynamically allocatable.

HUNK Short vectors, similar to LIST cells except that they have more than two components. This data type is fairly new and is still experimental.

Pointers

In MacLISP, as in most LISP systems, the unit of data is the pointer. A pointer is typically represented as a memory address, with the components of the data object pointed to in the memory at that address. The reason for this is that LISP data objects have varying sizes, and it is desirable to manipulate them in a uniform manner. Numbers, for example, may occupy varying numbers of words, and it is not always feasible to put one as such into the accumulators. A pointer, being only 18 bits, can always fit in one accumulator regardless of the size of the object pointed to; moreover, it requires only 18 bits for one data object to contain another, since it need actually only contain a pointer to the other.

Given a pointer, it is necessary to be able to determine what kind of object is being pointed to. There are two alternatives: one can either have a field in every data object specifying what type of object it is, or encode the type information in the pointer to the object. The latter method entails an additional choice: one can either adjoin type information to the memory address (in which case it takes more bits to represent a pointer), or arrange it so that the type is implied by the memory address itself (in which case the memory must be partitioned into different areas reserved for the various data types). MacLISP has generally used this last solution, primarily because of the half-word manipulation facilities of the PDP-10. Two memory address will fit in one word with no extra bits left over. (Contrast this with an IBM 370, which has 32-bit words and 24-bit addresses; on this machine one would use 32-bit pointers, encoding type information in the extra eight bits.) This is extremely useful because a list cell will fit in one word; the left half can contain a pointer to the CAR and the right half a pointer to the CDR.

The method MacLISP presently uses for determining the type of a data object involves using a data type table. The 18-bit address space (256K words) of the PDP-10 is divided into segments of 512 words. All objects in the same segment are of the same data type. To find the data type of an object given its address, one takes the nine high-order bits of the address and uses them to index the data type table (called ST, for Segment Table). This table entry contains an encoding of the data type for objects in the corresponding segment:

Bit 0	0 if atomic, 1 otherwise.
Bit 1	1 if list cells.
Bit 2	1 if fixnums.
Bit 3	1 if flonums.
Bit 4	1 if bignums.
Bit 5	1 if symbols.
Bit 6	1 if arrays (actually, array pointers; see below).
Bit 7	1 if value cells for symbols.
Bit 8	1 if number stack (one of bits 2-3 should also be set).
Bit 9	is currently unused.
Bit 10	1 if memory exists, but is not used for data.
Bit 11	1 if memory does not exist.
Bit 12	1 if memory is pure (read-only).
Bit 13	1 if hunks.
Bits 14-17	are currently unused.
Bits 18-35	(the right half) contain a pointer to the symbol representing the data type, namely one of LIST, FIXNUM, etc. The symbol RANDOM is used for segments containing no standard MacLISP data objects.

The encoding is redundant to take advantage of the PDP-10 instruction set and to optimize certain common operations. There is an instruction which can test selected bits in a half-word of an accumulator and skip if any are set. Thus, one can test for a number by testing bits 2, 3, and 4 together. Bit 0 (the sign bit) is 1 for list, hunk, and value cell segments (non-atoms) and 0 for all others (atoms). This saves an instruction when making the very common test for atom-ness, since one can use the skip-on-memory-sign instruction instead of having to fetch the table entry into an accumulator. The right half of a table entry contains a pointer to the symbol which the MacLISP function TYPEP is supposed to return for objects of that type. Thus, the TYPEP function need only extract the right half of a table entry; it does not

have to test all the bits individually. Finally, the system arranges for all the symbols to which a table entry can point to be in consecutive memory locations in one symbol segment. Since these symbols have consecutive memory address, the right half of a table entry can be used to index dispatch tables by type. For example, the EQUAL function, which determines whether two LISP objects are isomorphic, first compares the data types of its two arguments; if the data types match, then it does an indexed jump, indexed by the right half of a Segment Table entry, to determine how to compare the two objects.

By way of contrast, let us briefly consider the storage convention formerly used by MacLISP. Memory was partitioned into several contiguous regions, not all of the same size. The lowest and highest addresses of each region were known (usually the low address of one region was one more than the highest address of the region below it). To determine the data type of a pointer it was necessary to compare the address to the addresses of all the boundaries of the regions. This was somewhat faster than the current table method if only one or two comparisons were needed (as in determining whether a pointer pointed to a number, since the number regions were contiguous), but slower in the general case; furthermore, there was no convenient way to dispatch on the data type. On the other hand, the table method requires space for the entire 512-word table, even if only a small number of segments are in use. (There is another 512-word table for use by the garbage collector, the GC Segment Table (GCST), which doubles this penalty.) The deciding advantage of the table method is that it permits dynamic expansion of the storage used for each kind of data. The region method requires all list cells, for example, to be in a contiguous region; once this region is fixed, there is no easy way to expand it. Under the table method, any currently unused segment can be pressed into service for list cells merely by changing its table entry. An additional bonus of the table scheme is that the space required for the instructions to do a type-check is small, and so it is often worth-while to compile such type-checks in-line in compiled code rather than calling a type-checking subroutine.

In practice new data segments are not allocated randomly, but from the top of memory down. As new pages of memory are needed they are acquired from the time-sharing system and used for segments (on the ITS system, there are two segments per page). Compiled programs are loaded starting in low memory and working up; thus between the highest program loaded and the lowest data segment allocated there is a big hole in memory, which is eaten away from both ends as required. This hole has been whimsically named "the BIg Bag Of Pages" from which new ones are drawn as needed; hence the name "BIBOP" for the scheme. (The TOPS-10 timesharing system provided by DEC does not allow memory to be grown from the top down, but only from the bottom up. When running under this time-sharing system MacLISP has a fixed region for loading programs, and allocates new data segments from the bottom up.)

Data Representations

List cells, as mentioned above, are represented as single words. The CAR pointer is in the left half of the word, and the CDR pointer in the right half.

Fixnums are represented as single words which contain the PDP-10 representation of the number. As explained more fully in [Steele], this representation permits arithmetic to be performed easily. If a pointer to a fixnum is in an accumulator, then any arithmetic instruction can access the value by indexing off that accumulator with a zero base address.

Flonums are represented as single words in a manner similar to fixnums.

Bignums each have a single word in a bignum segment. The left half of this word is all zeros or all ones, representing the sign of the number. This representation of the sign is compatible with that for fixnums and flonums; thus the sign of any number can be tested with the test-sign-of-memory instruction. (Bignums were formerly represented as list cells with special pointers in the CAR; this did not permit the compatibility of sign bits, and made it difficult to test for either numbers or lists.) The right half points to a list of positive fixnums, which represent the magnitude of the bignum, 35 bits per fixnum, least significant bits first in the list. A list is used instead of a contiguous block of storage for both ease of allocation and generality of use. The least significant bits come first in the list to ease the addition algorithm.

Symbols are quite complex objects. Each symbol has one word in a symbol segment and two words in another segment. The right half of the one word points to the symbol's property list, which is an ordinary list; the left half points to the two-word block. These two words in turn are laid out so:

bits	0	pointer to value cell
"args" property		pointer to print name

The "bits" have various specialized purposes. The value cell for the symbol is in a value cell segment. Notice that bits 13-17 of the first word are zero, specifying no indexing or indirection. This permits an instruction to indirect through this word to get the value of the symbol. Getting the address of the two-word block also takes an instruction; thus one can get the value of a symbol in two instructions. The "args" property is used by the MacLISP interpreter for checking the number of arguments to a function (for symbols are also used to denote the names of functions). The print name is a list of fixnums containing the characters of the symbol's name, packed five ascii characters to the word.

The special symbol NIL is not represented in this manner. The address of NIL is zero. This allows a particularly fast check for NIL; one can use the jump-if-zero instruction. This is why accumulator 0 (which is also memory location 0) is reserved for NIL. Accumulator 0 normally contains zero itself; in this way taking CAR or CDR of NIL yields NIL. This allows one to follow a list by CDR pointers to a predetermined depth and not have to check at each step whether one has run off the end. (This trick was borrowed from InterLISP. [Teitelman]) Most functions make special checks for NIL anyway, so this non-standard representation is not harmful. PRINT, for example, just checks for NIL specially and just outputs "NIL" without looking for a print name. NIL does have a property list, but it is not stored where it is in other symbols; the property list functions must check for NIL (which takes only one instruction anyway). NIL has no value cell, and always evaluates to NIL.

One might wonder why normal symbols are divided up into two parts, and why the value cell is not simply part of the two-word block. The answer is that once constructed the two-word block normally does not change, and so may be placed in read-only memory and shared between processes. If several MACSYMA processes are in use, this sharing may ease core requirements by tens

of thousands of words.

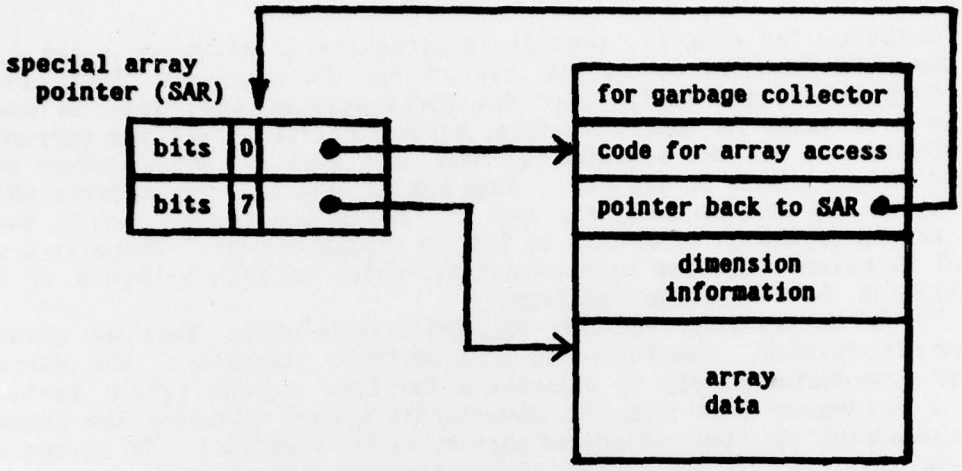
To save even more memory, symbols are not provided with value cells until necessary (most symbols are never actually given values). Instead, they are made to point to a "standard unbound" value cell, which is read-only and contains the marker specifying that no value is present. When an attempt is made to write into this value cell, the write is intercepted and a new value cell created for the symbol in question.

(Besides making parts of symbols read-only, MacLISP currently allows for read-only list cells, fixnums, flonums, and bignums. These are useful for constructing constant data objects which are referred to by compiled code but never modified, and for properties on property lists whose values are not expected to change (such as function definitions). In certain cases, such as the property-list modifying routines, checks are made for read-only objects, and such objects are copied into writable memory if necessary to carry out the operation. This copying causes the old read-only copy to be wasted from then on, but this is acceptable as such copying is seldom necessary in practice. This strategy may be contrasted to the approach of InterLISP [Teitelman], in which an entire page of memory is made writable if an attempt is made to modify any object on that page. This approach is more general than that of MacLISP, but in practice tends to reduce the sharing of pages among processes, increasing the load on the time-sharing system.)

Value cells, though not properly a MacLISP data type, are worthy of discussion. They are single words, containing a pointer in the right half and zero in the left half. This apparent waste of 18 bits is motivated by speed considerations. Compiled code often references the value cells of global variables. Since the left half of a value cell is zero, a test for NIL can be done with a single skip-if-memory-zero instruction; this is useful for switches. Furthermore, if a value cell is known to contain a list, the CAR or CDR can be taken in one instruction, using a half-word instruction with indirect addressing, because the index and indirection fields are zero, without having to fetch the value into an accumulator first. Similarly, if a value cell contains a number, the sign can be tested and the value (except for bignums) accessed by using indirect addressing. (It should be noted that compiled code does not keep local variable values in value cells, but uses even more clever techniques involving stacks.)

Arrays have a complicated representation because they can be of arbitrary size, and must be allocated as a contiguous block for efficient indexing. The solution chosen is to split it into two parts: a Special ARray cell (called SAR, not SAC, for some reason) in an array segment, and the block of data. The data itself is kept just below the hole in memory, floating above loaded programs. When new programs are loaded, the array data is shuffled upward in memory, and the special array pointers are updated. Similarly, when allocating a new array or reclaiming an old one it may be necessary to shuffle the array data.

The special array pointer is two words:



A complete discussion of the SAR contents and array access methods is beyond the scope of this paper. Notice, however, that the indirection and index fields are chosen to be 0 and 7 for the two SAR words. The first admits an indirection for calling the array as if it were a function, according to MacLISP convention; the second allows indexing off accumulator 7 for accessing the data from compiled code. See [Steele] for a fuller treatment of this.

Hunks are like list cells, but consist of several contiguous words. They are always a power of two in size, for convenience of allocation. Hunks of sizes other than powers of two are created by allocating a hunk of a size just big enough, and then marking some of the halfwords as being unused by filling them with a -1 pointer (actually 777777). This was chosen because it never points to a data object, and because it is easily generated with instructions that set half- or full-words to all ones. It is time-consuming to determine the actual size of a hunk, since one must count the number of unused halfwords, but then hunks were created as an experimental space-saving representation with properties somewhere between those of lists and arrays.

Garbage Collection

Every so often there comes a point when all the space currently existing for data objects has been allocated. At this point there are two alternatives:
[1] allocate a new segment for data objects of the type needed.
[2] attempt to reclaim space used by data objects which are no longer needed (by the process of garbage collection).
A study by Conrad indicates that the best strategy is to do [2] only if [1] fails because one's address space (256K words, in this case) is completely allocated, PROVIDED that one has the facility to compact one's data storage and de-allocate segments. [Conrad] Since MacLISP currently hasn't the ability to de-allocate segments ("once a fixnum, always a fixnum"), this strategy must be modified. One must be cautious about allocating a new segment, since the allocation cannot be undone; thus MacLISP tries garbage collection first unless explicitly told otherwise by the programmer, and then allocates a new segment if garbage collection fails to reclaim enough space for the required data type.

Suppose, for example, that it is necessary to allocate a new list cell. The CONS function checks the freelist for the data type "list cell"; if the freelist is not empty, then the first cell on that list is used. (There is a freelist for each data type, which consists of all the currently unused objects in all the segments for that data type, strung together such that each object points to the next. This can be done even for objects which ordinarily do not contain pointers, such as fixnums and flonums, since those objects are large enough to contain at least a single pointer. There is a set of fixed locations, one for each data type, which contain pointers to the first cells on the respective freelists.)

If, in our example, the list cell freelist is empty, then the garbage collector is invoked. Controlled by user-settable parameters, the garbage collector may decide simply to allocate a new list segment (which involves getting a new memory page from the time-sharing system, altering the Segment Table, and adding the newly allocated objects to the freelist). If it decides not to do this, or if the attempt fails for any reason, then the actual garbage collection process is undertaken. This involves finding all the data objects which are accessible to the user program. An object is accessible if it is pointed to by compiled code, if pointed to by a global variable or internal pointer register (such as accumulators 1-5), or if pointed to by another accessible object. Notice that this definition is recursive, and so requires a recursive searching of all the data objects to determine which are accessible. This searching is known as the mark phase of the garbage collector.

Associated with each data object is a "mark bit" for use by the garbage collector. As the garbage collector locates each accessible object, it sets that object's mark bit. For list cells, fixnums, flonums, bignums, and hunks, these bits are stored in a part of memory unrelated to the memory occupied by the data objects themselves. For each 512-word segment there is a "bit block" of 16 words, each holding 32 mark bits. The location of the bit block is found by using the top 9 bits of the address of the data object to index the GC Segment Table. (Bit blocks themselves are allocated in special "bit block" segments; thus bit blocks are treated internally as yet another data type. Occasionally the obscure error message "GLEEP - OUT OF BIT BLOCKS" is printed by LISP in the highly infrequent situation where it cannot allocate a new bit block after allocating a new segment which needs a bit block.) No bit blocks are needed for symbols and special array pointers. Recall that the left half of a symbol word points to a two-word block. Since such a two-word block is always at an even address, the low bit of the pointer to it is normally zero. This bit is used during garbage collection as the mark bit for that symbol. Special array pointers have room in them for a variety of bits, and one of them is used as a mark bit. Value cells are only reclaimed when the symbol pointing to them is reclaimed (and not even then, if compiled code points to the value cell, which fact is indicated by a bit in the two-word symbol block pointing to the value cell), and so they require no mark bits.

To aid the garbage collector in the mark phase, the GCST contains some bits which also encode the data type redundantly, in a form useful to the marking routine. The bits indicate whether the object must be marked, and if so the method of marking; they also indicate how many pointers to other objects are contained in the object now being marked.

After recursively locating and marking all accessible cells, the garbage collector then performs a sweep phase, in which every data object is examined, and those which have not been marked are added to the appropriate freelist. To aid the sweep phase, each GCST entry has a field by which all entries for segments of the same data type are linked together in a list. In

this way the garbage collector does not need to scan the entire segment table looking for entries for each type. For each segment, the garbage collector examines each data object in the segment and its mark bit, and adds the object to the appropriate freelist if the mark bit is not set. For symbols and arrays it also resets the mark bit at this time. (Bit blocks are reset at the beginning of the mark phase.)

If, in our example, the garbage collection process has not reclaimed enough list cells (as determined by another programmer-specified parameter), then it will try to allocate one or more new list cell segments. If, however, this causes the total number of list cells to exceed yet another programmer-specified parameter, then a "user interrupt" is signaled, and a function written by the programmer steps in. In MACSYMA, this function is the one that typically informs you:

```
YOU HAVE RUN OUT OF LIST SPACE.
DO YOU WANT MORE?
TYPE ALL; NONE; A LEVEL-NO. OR THE NAME OF A SPACE.
```

The reason for all these parameters is the necessary caution described above; if all the available segments get allocated as list cell segments (which can easily happen due to intermediate expression swell, for example), then they cannot be used for anything else, including compiled code. This is why, in MACSYMA, if you use up too much list space, you can't load up DEFINT thereafter!

Array data (as opposed to the SAR objects) is handled by a special routine that knows how to shuffle them up and down in core as necessary. When a new array is allocated, the garbage collector has the same decision to make as to whether to allocate more memory or attempt to reclaim unused arrays. The decision here is less critical, since memory allocated for arrays CAN be de-allocated, and so no programmer-specified parameters are used. Array data only goes away when the corresponding SAR is reclaimed by the normal garbage collection process (or when the array is explicitly killed by the user, using the *REARRAY function).

For the interested reader, the format of a GCST entry is shown here:

Bit 0	1 if data objects in this segment must be marked.
Bit 1	1 if this segment contains value cells.
Bit 2	1 if symbols.
Bit 3	1 if special array pointers.
Bit 4	1 if the right half of this data object contains a pointer (true of list, bignum, and hunk data objects).
Bit 5	1 if the left half of this data object contains a pointer (true of list and hunk objects -- note that symbols and special array pointers get special treatment). It is always true that bit 4 is set if this one is.
Bit 6	1 if hunks (in this case, the ST entry is used to determine the size of the hunk).
Bits 7-12	are unused.
Bits 13-21	contain the index into GCST of the next entry with the same data type, or zero if this is the last such entry. (Segment 0 never contains data objects, except NIL, which is treated specially anyway.)
Bits 22-35	contain the high 14 bits of the address of the bit block for this segment, if any.

Since bit blocks are 16 words long, the low four bits of the address of such a bit block are always zero. Thus the GCST entry only needs to contain the high 14 bits of the address. These 14 bits are right-adjusted in the GCST entry for the convenience of a clever, tightly-coded marking algorithm. This algorithm works roughly as follows:

[a] Shift the address of the data object to be marked right by 9 bits, putting the low 9 bits into the next accumulator.

[b] Use the high 9 address bits to fetch a GCST entry into the accumulator holding the high 9 address bits, skipping on the sign bit (whether to mark or not).

[c] Test bits 1, 2, 3 (special treatment), skipping if none are set.

[d] Shift the two accumulators left by 4 bits. This brings four of the low 9 address bits back into the first accumulator, which together with 14 bits from the GCST entry yield the address of a word in the bit block. The 5 bits remaining in the second accumulator indicate the bit within the word to use as the mark bit. Finally, bit 4 is brought into the sign bit of the first accumulator.

[e] Rotate the second accumulator, bringing the 5 bits to the low end.

[f] Indexing off the first accumulator, fetch the word of mark bits.

[g] Set a mark bit in the word, skipping if it was not already marked. (If this doesn't skip, then we exit the marking algorithm. It is not necessary to store back the word of mark bits.) The bit is selected by indexing off the second accumulator into a table of words, each with one bit set.

[h] Store back the word of mark bits.

[i] Test the sign bit of the first accumulator (bit 4 of the GCST entry), jumping to the exit if not set.

[j] If bit 1 is set (bit 5 of the GCST entry), recursively mark the pointer in the left half. If bit 2 is set (bit 6 of the GCST entry), mark all the pointers in the hunk.

[k] Iteratively mark the pointer in the right half.

I have taken the trouble to outline these steps carefully because most of them are single PDP-10 instructions, carefully designed to perform two or three useful operations simultaneously. The point is that the careful design of tables and the use of redundant encoding can greatly increase the speed of critical inner loops. (It should also be mentioned that such careful thought about design is usually warranted only for critical inner loops!) I should also mention that most of the constants which have been mentioned in this paper (bit numbers, sizes of segments, and so on) are represented symbolically in the text of the MacLISP code; one can change the size of a segment by changing a single definition, and the sizes of fields in GCST entries, positions of bits, and so on will be adjusted by assembly-time computations. I have used numbers in this paper only for concreteness.

For certain spaces the mark bits are actually used in the inverted sense: 1 means not marked, and 0 means marked. This allows the sweep loop to test for an entire block of 32. words all being marked by testing for a zero word of mark bits; the loop can then just skip over the block, and avoid testing the individual bits. The test for a zero word is done while moving the word into an accumulator, which has to be done anyway, and so is essentially free.

The Address Space Problem

One of the difficulties currently facing MacLISP is the "limited" address space provided by the PDP-10. The architecture of the machine inherently limits addresses to 18 bits; hence a single program cannot address more than 256K words of memory. Combined with the fact that MacLISP does not presently allow for de-allocation of data segments (or of loaded compiled code, for that matter), this severely limits the use of memory. Some MACSYMA problems, for example, would require much more than 256K of programs and list data to solve; others require less than 256K at any one time, but cannot be run because of the de-allocation difficulty.

It is fairly clear that completely solving the de-allocation problem would be more trouble than it is worth, and would not stave off the fundamental difficulty indefinitely. As both MACSYMA problems and MACSYMA itself grow in size, we will feel more and more the "address space crunch". The only general way to solve this problem is to arrange for a bigger address space.

There are three solutions which are presently at all realistic. Two involve continued use of the PDP-10 architecture, but modified in several ways to allow programs to access more memory. These modifications may or may not be made available by DEC, and may or may not be retrofittable to the MACSYMA Consortium KL10 processor. The difference between the two schemes involves the decision as to whether MacLISP data pointers should still fit into 18 bits. If not, there is immediately a factor-of-two memory penalty, since list cells must be two words instead of one. However, there are also some technical advantages to such an arrangement, as well as the obvious advantage that list space can become bigger than 256K. If pointers are kept to 18 bits, then all LISP data must fit in 256K, but any amount of compiled code and any number of arrays could be loaded. Both of these schemes have been worked out on paper to a great extent by Guy L. Steele Jr. and Jon L. White, to compare their merits and to prepare for the possibility that one of them may be needed. Either scheme would require a good deal of work (at least one to two man-years) to implement fully in both the interpreter and the compiler.

The third solution involves moving to another machine architecture altogether. This leaves open the choice of machine. Few commercially available machines are as conducive to the support of LISP as the PDP-10, and it probably would not be practical to undertake a completely new implementation. MacLISP does presently run on Multics (on a Honeywell 6180 processor), but is rather slow, and the Multics system is expensize and not widely available. The best bet in this direction seems to be the LISP machine, designed by Richard Greenblatt, Tom Knight, et al. at the MIT Artificial Intelligence Laboratory. The prototype machine has been working for a number of months now, and the basic software is beginning to show signs of life. It is not inconceivable that MACSYMA may be run experimentally on it by summer 1977. The LISP machine has a 24-bit address space, and makes more efficient use of its memory than even the PDP-10. However, although it is much less expensive than a KL10, it is not designed for time-sharing.

The PDP-10 implementation of MacLISP and of MACSYMA will certainly be useful for at least the next five to ten years. After that, only time can tell.

Summary

MacLISP is designed to be an efficient, high-level systems programming language, rather than primarily an applications programming language. Its internal organization is a carefully chosen balance between useful generality and special-case efficiency tricks. A thoughtful choice of data and table representations can exploit the architecture of the host machine to gain speed in critical places without great loss of generality. The use of symbolic assembly parameters can avoid tying the system to a single rigid format. The greatest effort has been expended on speeding up type-checking, access to values in global variables, and garbage collection, since these are among the most frequent of LISP operations. The address space crunch may eventually force yet another redesign if the PDP-10 architecture is retained.

References

[Steele] Steele, Guy L. Jr. "Fast Arithmetic in MacLISP." Proceedings of the MACSYMA Users' Conference, NASA CP-2012, 1977. Also MIT AI Memo 421 (Cambridge, August 1977).

[Teitelman] Teitelman, Warren. InterLISP Reference Manual. Revised edition. Xerox Palo Alto Research Center (Palo Alto, 1975).

[Conrad] Conrad, William R. A compactifying garbage collector for ECL's non-homogeneous heap. Technical Report 2-74. Center for Research in Computing Technology, Harvard U. (Cambridge, February 1974).