BOLT BERANEK AND NEWMAN INC

CONSULTING · DEVELOPMENT · RESEARCH

AFCRL-66-426

STORAGE MANAGEMENT IN LISP

Daniel G. Bobrow

Bolt Beranek and Newman Inc

50 Moulton Street
Cambridge, Massachusetts 02138

Contract No. AF19(628)-5065
Project No. 8668

Scientific Report No. 4

(The work reported was supported by the Advanced Research Projects Agency, ARPA Order No. 627, dated 9 March 1965)

June, 1966

Prepared
for

AIR FORCE CAMBRIDGE RESEARCH LABORATORIES
OFFICE OF AEROSPACE RESEARCH
UNITED STATES AIR FORCE
BEDFORD, MASSACHUSETTS

AFCRL-66-425

STORAGE MANAGEMENT IN LISP

Daniel G. Bobrow

Bolt Beranek and Newman Inc

Bolt Beranek and Newman Inc
50 Moulton Street
Cambridge, Massachusetts 02138

Contract No. AF19(628)-5065

Project No. 8668

Scientific Report No. 4

(The work reported was supported by the Advanced Research Projects Agency, ARPA Order No. 627, dated 9 March 1965).

June, 1966

Prepared for

for

AIR FORCE CAMBRIDGE RESEARCH LABORATORIES
OFFICE OF AEROSPACE RESEARCH
UNITED STATES AIR FORCE
BEDFORD, MASSACHUSETTS

ABSTRACT

Storage allocation, maintenance, and reclamation are handled automatically in LISP systems. Storage is allocated as needed, and a garbage collection process periodically reclaims storage no longer in use. A number of different garbage collection algorithms are described. A common property of most of these algorithms is that during garbage collection all other computation ceases. This is an untenable situation for programs which must respond to real time interrupts. The paper concludes with a proposal for an incremental garbage collection scheme which allows simultaneous computation and storage reclamation.

SECTION I

INTRCDUCTION

Storage allocation, maintenance, and reclamation are handled automatically in LISP systems. Storage is allocated as needed, and a garbage collection process periodically reclaims storage no longer in use. This paper describes the storage allocation process and a number of different garbage collection algorithms. A common property of most garbage collection algorithms is that during collection all other computation must cease. This is an untenable situation for programs which must respond to real time interrupts. The paper concludes with a proposal for an incremental garbage collection scheme which allows simultaneous computation.

SECTION II
STORAGE ALLOCATION

In any LISP system there are a number of different types of storage that are used. These storage types may be classified in two different ways. The first is by content: pointers, absolute quantities, and numbers relative to some origin. List pointers, floating point numbers and computer instructions in relocatable code are examples of these three types respectively. The second classification is by function, the use of storage in the system as: pushdown list, list structure (2 pointer quantities); single word blocks for numbers; and binary programs and arrays. Figure 1 shows a typical LISP storage map by function of the space used. Both of these categorizations, by content and function, will be useful in considering the problems of storage management in LISP.

The philosophy in the LISP system is that the user never need concern himself with storage allocation; it should be handled automatically. Each function as it is called has access to the pushdown list (PDL) for temporary storage. Management of this PDL is in the hands of the interpreter, for interpreted functions, and in system subroutines invisible to the user for compiled programs. Similarly, the compiler knows how to grab space for code for any function it compiles. However, unlike the PDL, once storage has been allocated for compiled code, this space assignment is permanent.* The area labelled free

* This is not strictly true but is close enough for purposes of this discussion.

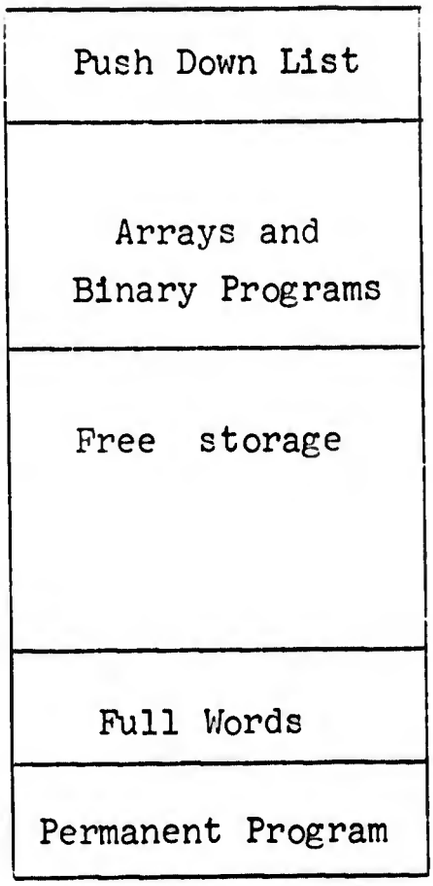

Fig. 1: Typical Storage Areas in LISP

storage on Figure 1 initially contains a single list of all list pointer storage available. As list cells are required, they are removed from the free storage list. There are only two functions in LISP which can acquire an element from free storage, cons and ratom.* The function cons adds an element to the beginning of a list, and ratom reads in an atom, and creates a special list structure for each new atom. If x = (A B) and y = (D C E) then the form z = cons[x ; y] has the value ((A B)D C E), where a pointer to (A B) has been placed in the car pointer of the new list element z, and a pointer to (D C E) has been placed in the cdr pointer of z. Thus the cdr pointer is a link to the rest of the list past the first element, and the car pointer points to the first element of the list.

The function ratom is used in the read program of LISP to create a special type of list called an atom. A list which is associated with an atom is distinguished within some LISP systems by having a special mark as the first element of this list. In others the list associated within any atom is distinguished because it begins in a reserved block in the free storage area. We will talk about atoms as marked in the first way, although the two are equivalent.

There are two types of atoms, literal and numerical. Numerical atoms are distinguishable from literals. Externally a literal atom is any string of letters or digits starting with a letter, and delimited on either side by a left parenthesis, right parenthesis, period, space, or comma. All other delimited strings of letters or digits are converted to numbers internally

---

* This is not strictly true but is close enough for purposes of this discussion.

(if possible) and stored in a word in full word space. Then a word for such an atom head is taken from the free storage list, an atom mark is put in the car pointer, and a pointer to this full word cell is placed in the cdr pointer of this atom head.

A literal atom (but not a numerical atom) is uniquely determined by the string of letters and digits in its external representation. Before a new cell is created for this atom, a search is made through all atoms currently in the system. If an atom is found which has the same p-name (print name) as the atom just read in, then a pointer to the previous atom head is used. If no identical atom is found, then a cell is taken from free storage, and marked as an atom head. Then the p-name of the atom is stored associated with the atom. In some LISP systems a pointer to this p-name is stored in a hidden cell for this atom (a cell only accessible to the print program). In other systems, the p-name is placed on the property list of the atom following a flag, the atom PNAME. (The property list of an atom is the list pointed to by the cdr pointer of the atom head.)

Since literal atoms are unique, they can be used to identify uniquely a function and/or a variable of a function. In some systems, special cells associated with the atom are used to store a variable value and a pointer to a function definition when these are identified with this atom. In others these two properties are found on the property list of the atom. In compiled functions, values of variables are usually placed on the pushdown list. Since the property list of an atom is available to the programmer, he too may place on this property list any properties which he wants uniquely and immediately associated with a given literal string (atom). We will not discuss further here the use of these property lists in programming, but many uses should be obvious.

Let us note in passing that the search for atoms is made relatively efficient by a hash-coding technique. One commonly used technique is to hash the literal string into a 7 bit quantity (7 is used only for illustrative purposes, and could be any number - 7 is used in LISP 1.5 on the 7094). This 7 bit quantity selects one of 128 lists of atoms, and only this selected list is searched for the atom in question. Thus only 1/128 of the total number of atoms in the system need be searched. This technique is often referred to as a bucket sort.

In systems which have arrays and compiled functions, blocks of storage have leaders which contain, explicitly or implicitly, the following information:

1) Type of block
2) Length of block
3) Position of quantities which are relative to beginning of block
4) Position of pointer quantities
5) Starting address of block.

Not all of the systems provide all of the information, but the implications of providing each will become clear in the discussion below on garbage collection.

SECTION III

RECLAMATION

When storage of any kind is exhausted, the system initiates a process known as garbage collection, to reclaim all storage that is no longer in use. Other work in the system stops while garbage collection is in progress. Note that the user never requests anything be returned to free storage, and thus garbage will accumulate. Before going on to discuss the garbage collection process itself let us briefly mention the arguments for garbage collection as opposed to explicit restoration by the programmer, aside from the fact that it is obviously easier for the programmer.

In LISP, many lists can have a single common sublist. Thus when a programmer erases a list and returns it to free storage he must be sure that it is not still being referenced in any other list. This problem can be ameliorated, as in SLIP, by the use of reference counters. In this scheme, a reference count is incremented every time a list is referenced in any structure. Then the erase command merely decrements the count of every substructure in a structure. Only when its reference count reaches zero is a list actually returned to free storage.

However, this scheme runs into another more serious problem. In LISP we can construct circular lists, lists which reference themselves, perhaps many times. In one large project at Bolt Beranek and Newman, this is a critical property of the

structures used. In this case, the reference count will never reach zero, but may remain at an arbitrarily large number indicating the number of self references, and the list will never be really erased. There are solutions to this problem too. They involve searching a list for self references or keeping separate self referenced counters. This becomes awkward, and in addition, reference counters take space and space is often at a premium. Thus, because of the obvious convenience to the programmer, and the awkwardness of alternatives, LISP uses garbage collection.

The basic garbage collection process consists of two phases, marking and collection. The basis of the marking phase is a set of the beginnings of all lists used. In most LISP systems this set consists of:

1) All atom heads
2) Pointers on the currently active pushdown list
3) Pointer quantities in arrays and programs.

From each pointer, every list is traced, and every cell accessible from any list is marked as in use. If a marked cell is seen in tracing through a list, tracing in that direction terminates since obviously that sublist and all of its substructures have already been marked.

When this marking process is complete, collection of garbage begins. A linear sweep is made through memory, and all unmarked cells are placed on a new free storage list. Unmarked full works are placed on a separate free storage list of their own, or otherwise marked as available. Unused array space and compiled code space is also marked as available (in some systems). All marked cells are unmarked and the user continues his process where he left off, with new space

available if he indeed had any garbage around. Of course, if all the space is being used the system chokes to death.

This is the garbage collection algorithm used in LISP 1.5 on the IBM 7094 at MIT. What are the problems with this algorithm? First note that the garbage collector must be able to recognize which items on the pushdown list are pointers, and which are not. Thus recognition of cells by content is imperative. For example, there will be returns from subroutines which appear on the PDL. One way recognition is facilitated is by making non-pointers unique quantities which cannot be pointers. Another is to put unique flags on the PDL which will identify non-pointers. In LISP II, a more sophisticated scheme is used. Associated with each cell is a map (constructed at compile time) which indicates the nature of each cell on the PDL in the neighborhood of a call, at the time of the call.

In tracing down each list, the tracing program can either follow the car pointer or cdr pointer first and so on down the line. In either case the other pointer must be saved on a pushdown list. This pushdown list must be long enough to hold all the alternate pointers in the longest car-cdr chain in any list structure. This is usually no problem, but if there are unusually long lists this PDL may be exhausted. There is an algorithm due to Peter Deutsch which allows marking to proceed without using a pushdown list. We won't describe it in detail here but it modifies the list structure itself to point to structures which should be marked but aren't yet. In addition to the marking bit, it utilizes another bit to indicate if the car pointer has been marked yet. One might describe the process as turning the list structure inside out and marking it from the bottom up.

An alternative to this scheme is to use a bit table, with one bit for each word to indicate which cells are still to be marked. This has some advantages in a LISP in a paging environment. Note that a bit table or a bit in each work must be utilized to indicate which words are marked. Again the bit table may be advantageous in a paging environment, if the table is always around during garbage collection, since the garbage collector can determine whether a cell is marked without having to get the word, and perhaps reference a page not in core.

Another problem with the garbage collection scheme presented above is that no atom heads are ever collected. This can be circumvented by marking only those atoms with properties (other than a PNAME, which all atoms have) or which are referenced in some list. Still another problem is the fact that if array or program space is made available, the blocks that are available may not be contiguous and none may be large enough for the next needed allocation. This is closely related to the problem of reassigning the boundaries of each area of storage, as shown in Figure 1.

SECTION IV

COMPACTING GARBAGE COLLECTION

With a garbage collector which compacts each area of storage, we remove the problems of moving boundaries between regions of storage and obtaining large enough array or program blocks. In a compacting garbage collector, there are four phases:

1) Marking
2) List structure compacting and array relocation computation
3) Address adjustment
4) Relocation of arrays, collection of unused list cells, and unmarking of used cells.

The marking phase is essentially the same as described above. List structure compacting is accomplished by a folding operation (an idea first suggested by D. Edwards). A pointer T is set to the top of the free storage area, and a pointer B is set to the bottom of free storage. We will talk of compacting toward the bottom of free storage. T is moved down until the first marked cell is encountered. Then B is moved up until the first unmarked cell is encountered and the contents of the T-cell are inserted into this B-cell. In the T-cell we insert the address of the B-cell, i.e., the new address of this structure. This process is iterated until the T pointer and B pointer are identical. At this point all list cells are below this location T, and any pointers to cells above T need to be adjusted in the next phase. Full word compacting is done similarly.

Array relocation computation (this includes compiled functions as a special case), is done as follows. Array space is swept and as each marked array is encountered, its computed new position is entered into its header in the cell for start location of this block. This new start address can be computed from the starting address of array space and the lengths of other marked arrays which will be below the current array in array space. The arrays are not moved at this time because this might cause an array cell to overlap a relocation address.

In phase 3 all address adjustments are made. Any list pointer above T is changed to the new pointer found in the cell pointed at. Similarly any pointer to an array is changed to a pointer to the new location of the array.

In phase 4, all the cells above T are free storage. They need not even be linked into a list since this area is compact, and can be accessed by indexing. Arrays are relocated to their new position, starting at the bottom, with any necessary relocation of internal items. There is no problem of overlapping. The contiguous block above the last array is now available for allocation for new arrays. All marked structures are now unmarked. Since all items in storage can be relocated, boundaries between storage areas can be changed in this process.

Minsky has suggested an alternative approach for compacting garbage collection. It requires the use of a serial secondary storage medium. It essentially puts out a symbolic transcription of internal storage, with the additional property that common substructures are retained. It also has the property that cdr chains usually become linear sequences. This is most advantageous in paged computer memories.

SECTION V

INCREMENTAL (REAL TIME) GARBAGE COLLECTION

If a LISP system is used for computations which are keyed to real time, then the garbage collection schemes presented above are inadequate. By keyed to real time we mean that a LISP computation must be made within milliseconds of an event in the outside world. However, all computation stops when a garbage collection is in progress and even on a 7094 with only 10,000 words of free storage, garbage collection takes about half a second. Therefore, a LISP system could not react fast enough to outside interrupts while garbage collecting. As a graphic example, consider a LISP pattern recognition program looking at a ping pong ball and directing an arm to hit the ball with a paddle. It would lose a point on every garbage collection. The following modifications to the compacting garbage collection scheme above will, I believe, allow simultaneous incremental garbage collection and real time computation. It is assumed that the real time process (or any other operating process) is not consuming storage so fast that the garbage collector will not have time to finish.

Garbage collection is initiated when only a certain small fraction (say 10%) of storage is left. It is a separate process and shares central processor time with other operating LISP processes as is standard in time sharing (multiprogramming) systems.

The marking process proceeds as usual, with marking done on the pushdown list from older items to new. When the garbage collector reaches the current bottom of the pushdown list, marking is complete. However, during the marking phase any process going on simultaneously must take special precautions. On any writing done in the process, specifically on rplaca's, rplacd's, and set's with prog's, the system must check to see if the changed cell has already been marked. If so, the item entered into this marked cell must be placed on the pushdown list of the garbage collector, unless, of course, this new item has also been marked.

During the list structure relocation phase and the list cell adjustment phase, any references through pointers (car's and cdr's) must be checked to see if they refer to a point above the current T pointer. If so, then an indirect address must be computed, the right relocated cell obtained. Address adjustment for arrays and relocation of arrays must go on simultaneously. A special cell for the array currently being moved must be provided, and a special computation must be made to find an element of the array being moved.

The incremental garbage collector described here is currently being implemented for a LISP 1.5 system on an SDS 940 computer. It is hoped that this system will allow simultaneous use by a number of users, including some real time processes, with only a tolerable loss of speed, not a cessation of computation, during garbage collection.

REFERENCES

1. McCarthy, J., et al LISP 1.5 Programmers Manual MIT Press, Cambridge, Mass.; 1964.
2. Berkeley, E. and Bobrow, D. G. (editors) The Programming Language LISP: Its Operation and Application, MIT Press, Cambridge, Mass., 1966.
3. Edwards, D. J. LISP II Garbage Collector, Memo 19, Artificial Intelligence Group, MIT; 1962.
4. Minsky, M. L. A LISP Garbage Collector Algorithm Using Secondary Serial Storage (revised), Memo 58, Artificial Intelligence Group, MIT; 1963.
5. Bobrow, D. G. et al, The BBN LISP System, BBN Report 1346, 1966.
6. Bobrow, D. G. et al, LISP in a Paging Environment, BBN Report in preparation; 1966.

BLANK PAGE

Unclassified
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Massachusetts 02138

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE

STORAGE MANAGEMENT IN LISP

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Scientific Report #4 - Interim

5. AUTHOR(S) *(Last name, first name, initial)*

Bobrow, Daniel G.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
June, 1966	16	6

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
Contract No. AF19(628)-5065	BBN Report #1423
b. PROJECT AND TASK NO. 8668	
c. DOD ELEMENT 6154501R	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d. DOD SUBELEMENT None	AFCRL-66-426

10. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
The work reported was supported by the ARPA, P. R. No. CRI-56176, ARPA Order No. 627 dated March 1966	Air Force Cambridge Res.Labs. Office of Aerospace Res.(CRB) USAF, Bedford, Massachusetts

13. ABSTRACT

Storage allocation, maintenance, and reclamation are handled automatically in LISP systems. Storage is allocated as needed, and a garbage collection process periodically reclaims storage no longer in use. A number of different garbage collection algorithms are described. A common property of most of these algorithms is that during garbage collection all other computation ceases. This is an untenable situation for programs which must respond to real time interrupts. The paper concludes with a proposal for an incremental garbage collection scheme which allows simultaneous computation and storage reclamation.

DD FORM 1 JAN 64 1473

Unclassified
Security Classification

Unclassified

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
LISP List Processing Storage Management Storage Allocation Real Time Garbage Collection Garbage Collection Free Storage Maintenance						

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2a. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2b. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year, or month, year. If more than one date appears on the report, use date of publication.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8a. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8b, 8c, & 8d. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9a. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9b. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through _____."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through _____."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through _____."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS), (S), (C), or (U).*

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional.

Unclassified

Security Classification