19 AFCRL- 66-774

060
100

THE STRUCTURE OF A LISP SYSTEM USING TWO-LEVEL STORAGE

Daniel G. Bobrow
Daniel L. Murphy

Bolt Beranek and Newman Inc
50 Moulton Street
Cambridge, Massachusetts

Contract No. AF19(628)-5065

Project No. 8668

Scientific Report No. 6

4 November 1966

(The work reported was supported by the Advanced Research Projects Agency, ARPA Order No. 627, Amendment No. 2, dated 9 March 1965)

Prepared For:

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

AFCRL- 66-774

THE STRUCTURE OF A LISP SYSTEM USING TWO-LEVEL STORAGE

Daniel G. Bobrow
Daniel L. Murphy

Bolt Beranek and Newman Inc
50 Moulton Street
Cambridge, Massachusetts

Contract No. AF19(628)-5065

Project No. 8668

Scientific Report No. 6

4 November 1966

(The work reported was supported by the Advanced Research Projects Agency, ARPA Order No. 627, Amendment No. 2, dated 9 March 1965)

Prepared For:

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

TABLE OF CONTENTS

FIGURES

ABSTRACT

In an ideal list-processing system there would be enough core memory to contain all the data and programs. This paper describes a number of techniques used to build a LISP system which utilizes a drum for its principal storage medium, with a surprisingly low time-penalty for use of this slow storage device. The techniques include careful segmentation of system programs, allocation of virtual memory to allow address arithmetic for type determination, and a special algorithm for building reasonably linearized lists. A scheme is described for binding variables which is good in this environment and allows for complete compatibility between compiled and interpreted programs with no special declarations.

SECTION I

INTRODUCTION

LISP is a list-processing language which is being used extensively in research in artificial intelligence. In the ideal list-processing system, there would be enough core memory to contain all the data that were to be referenced over a long time period. In this case, a data reference would take on the order of one or two microseconds, a speed typical of present core memories. When large systems are constructed, requiring upwards of one hundred or two hundred thousand words, the cost of core memory usually becomes prohibitive. Memory size requirements of this order of magnitude are, however, not the exception buy rather the rule for many projects planning research on natural language, speech processing, and a host of other areas.

Thus it becomes necessary to consider the use of bulk storage memory devices such as magnetic drums and discs. The problem in using such devices is that, while their data transfer rate is sufficiently high (5-10 μsec/wd), one must wait for the data on the rotating medium to come around to the read position. The average time to access data is one-half the rotation time, or typically about 17 to 33 milliseconds. To use a drum, then, as if it were core memory, i.e. to read single words as they are required, would increase data reference time by a factor of around 20,000 making any list-processing system uselessly slow for practical problems.

As a first step toward utilizing drum storage efficiently, we organize the drum into blocks of words, or pages, and bring into core an entire block of words whenever one from that block was required. As may be seen from the timings above, the extra time required to transfer 200 to 300 words instead of one, is negligible compared to the access time. If multiple references are made to a block once it has been moved into core, then the speed of data references is increased by a factor equal to the number of references made to a given page before another must be brought in.

In this paper we describe a number of techniques we have used in the BBN LISP system to maximize the number of in-core references per drum reference. These techniques should increase the speed of operation of any list-processing system embedded in a time-sharing system which uses paging to map a large virtual memory stored on a drum into a smaller core memory, e.g. the MULTICS system (3).

SECTION II

ORGANIZATION OF CORE MEMORY

Our LISP system has been implemented on a Digital Equipment Corporation PDP-1. This machine has a core memory of 16K (K = 1024) and a drum memory of 88K. Access time to one 18 bit word of core is 5 microseconds, and average access time to a word on the drum is 17 milliseconds. The PDP-1 has no index registers, floating point instructions, special push-down-list instructions, or paging hardware. It has an unusually large collection of I/O devices, including paper tape reader and punch, teletypes with reader and punch, mag-tape drives, high speed display and light pen, and a Graphicon (RAND) tablet. The LISP system can communicate with all these I/O devices.

The BBN LISP system contains both an interpreter and a compiler. The operation of the compiled code is completely compatible with the interpretation by the interpreter of S-expression definitions of functions. One may run mixed sets consisting of functions which are interpreted and others which are run after being compiled. The scheme for binding variables which allows this complete compatibility is discussed in detail below.

The 16K of core memory must be allocated among permanent code, list structure storage, and compiled code. To this end we have dedicated 4K to compiled code, 4K to the

supervisor and permanent code, and 8K to list structure and pushdown list storage. Since a list element consists of two 18 bit words, this 8K of memory is equivalent to at most 4K LISP words.

Segmentation of System Code

The permanent code for the system is well over 10K in length itself. In order to stay within the allotted 4K of memory we have segmented the system code into 6 overlays which have minimal interaction between them, minimizing the number of swaps necessary between overlays. We keep permanently in core the interrupt routines for servicing user on-line interaction, and an elementary time-sharing supervisor (our system allows a small number of users - usually 2 or 3 - to use the machine as a LISP dedicated time-sharing system). The remainder of the 4K is used as swapping area for the following overlays:

1. Interpreter and compiled code runner
2. Some non-critical special machine coded subroutines for manipulating list structure
3. Input-Output and formatting
4. A special package for manipulating the rather stupid magnetic tape drives on the machine
5. The garbage collector
6. An initialization package

Since all segmentation and paging is done by software on this machine, we felt that explicit segmentation and complete swapping of overlays was preferable to calling subroutines off the drum as needed. This is in contradistinction to our philosophy on running compiled LISP code. For the system code we know what reasonable segments are; in the latter case the system could not know (and we did not want

to put the onus on the user for keeping track of) segments that would fit in 4K of storage. All overlays are absolute code and can therefore run only in fixed locations.

Compiled LISP Functions

Programs compiled from S-expression definitions of LISP functions are stored on the drum in relocatable form. When running, these functions are contained in a ring buffer in core of about 3400 words, properly relocated. Let us consider what happens when a compiled function is called. The call contains a pointer to the atom which names the function (essentially the symbolic name of the function). This name is used to search an in-core transfer vector containing starting locations of all routines in core. The search is done by "hashing" the name, and searching the transfer vector until the name or an empty space is found. We have found that with the ring buffer full, the transfer vector of 128 words (64 name-starting location pairs) is only about 1/4 full. Therefore the average number of checks to determine the presence of a function in core is only slightly larger than 1. If the function called is in core, a transfer is made to the starting address given. Thus, in this case, we have made no drum references to link functions. This procedure could be improved if we wanted to modify the code of the calling function to contain a jump indirect through the transfer vector. However, we prefer that the only address binding done on compiled code be simple relocation.

If the function called is not in core, we obtain its drum address from the function cell of its atom. We then read into the ring buffer the first page of the compiled code for this function (i.e. from its initial position to the end of the 256 word block which constitutes a page on the drum).

The first word of the program contains the length of the program. Successive pages are read in until the entire program is in core. If at any time we overlap the end of the ring buffer, we start again from the beginning of the ring buffer with the beginning of the program. The transfer vector is updated by removing entries for functions which have been wholly or even partially overwritten, (taking care to mark them properly for the hash lookup) and a new entry is added for the program. Returns from calls are also made through the transfer vector. Thus, programs which have called subroutines may be overwritten. In this case, the program will be recalled from the drum before the return is effected. Sets of functions seem to stabilize under this system, and if all the programs to be used for a reasonable period can fit in core, they soon reside there.

SECTION III

ORGANIZATION OF THE DRUM MEMORY

LISP assumes that it is operating in an environment containing 128K words, that is from 0 to 400,000 octal. Only 88K actually exist on the drum. The remaining portion of the address space is used for representation of small integers between -32,767 and 32,767 (offset by 300,000 octal), as described below. All data storage is contained within this virtual memory, including list structure, compiled code, atom value cells, property cells and function cells, print name storage and pushdown list storage. This virtual memory is divided into pages of 256 words. Reference to the virtual storage are made via an in-core map which supplies the address of the required page if it is in core, or traps to a supervisory routine if the page is not in core. This drum supervisory routine selects an in-core page, writes it back on the drum if it has been changed, and reads the required page from the drum. Closed subroutine references to an in-core word through the map takes approximately 170 microseconds (because of the poor set of operation codes on the machine, and the lack of an index register). A reference to a word not in core, which must be obtained from the drum, takes between 17 and 33 milliseconds. It takes the longer time if a page must be written out on the drum before the referenced page can be read in. Thus, it really pays to minimize drum references.

Type Determination of Pointers

In standard wholly-in-core LISP systems the type of element a pointer is referencing (for example, an atom versus an S-expression) can only be determined by looking at the item itself, which might require a drum reference. We avoid this unnecessary drum reference by dividing the virtual memory space into a number of areas as shown in Fig. 1. As can be seen from this map of storage, simple arithmetic on the address of a pointer will determine its type. We chose to allocate storage rather than provide an in-core map of storage areas, because the map would take up valuable in-core space, and every additional page of storage that we can squeeze into core reduces the number of drum references.

Literal Atoms

When a string of characters representing a literal atom is read in, a search is made to determine if an atom with the same print-name has been seen before. If so, a pointer to that atom is used for the current atom. If not, a new atom is created. Thus, as in all LISP systems, a literal atom has a unique representation.

Four cells (words) are associated with each literal atom. These cells contain pointers to the print-name of the atom, the function which it identifies, its top level or global value, and its property list. A pointer to an atom points to its value cell. Since these value cells occur in only one part of the address space, one can tell from a pointer (address) whether or not it is pointing to a literal atom.

Instead of having the other cells on the same page with the value cell, each is put in a separate space in a position computable from the address of the value cell. Separating value

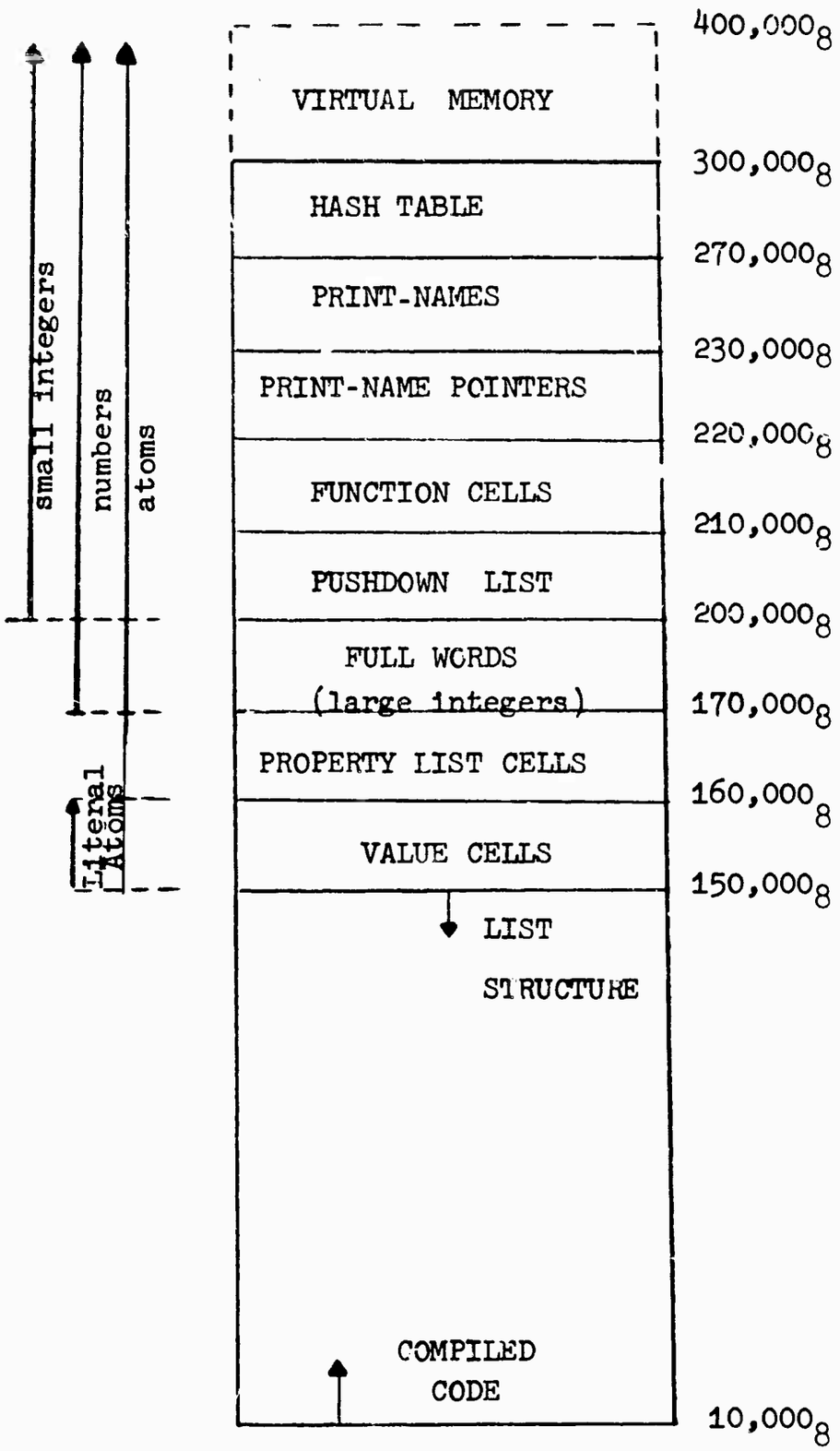

Fig. 1. Organization of the Virtual Memory.

cells and function cells, for example, is useful because most users will not use the same name for a global variable as they will for a function, and, therefore, if the four cells are brought in whenever any one was asked for, it is likely that the other three cells would never be referenced. Yet, they use up room in core which could be used for other storage. Similarly, the print-name pointers associated with atoms are needed during input and output, but rarely during a computation. Therefore, during computation these cells are never in core.

Numerical Atoms

In 7094 LISP, numerical atoms (numbers) do not have a unique representation; that is, a number of different pointers may be referencing numbers with the same value. This implies that for comparison of numbers, or for arithmetic operations, the value of the numbers must be obtained, and comparison of numerical atoms cannot be just a comparison of pointers.

The values of numbers are stored in "full word" space. In 7094 LISP, pointers to numbers in list structure do not point directly to the values of the numbers in full word space. For each numerical atom, a special word from free storage is acquired with bits set in this word to indicate type, and a pointer to the value of the number in full word space. A pointer in list structure which references a number points to this "header" word. Because type information is implicit in pointers (addresses) in our LISP system, we do not need this extra level of indirection, and pointers to number values directly address free word space. This obviously saves possible drum references in arithmetic operations and comparisons.

In addition, we utilize the fact that not all addresses in the 17 bit address space of the drum can legitimately appear as

pointers in list structure. Pointers between 200,000 (octal) and 400,000 (octal) are, therefore, used in the context of list structure to represent directly "small" integers between -32767 and +32767 (offset by 300,000 octal). Thus, a pointer of 300,003 (octal) occurring in a list is the number 3. Again, eliminating a level of indirectness reduces the number of drum references required. Another advantage of using this form for small integers is that numbers in this range are represented uniquely; thus, arithmetic comparisons can be done directly on the pointers. In addition, no additional storage is required. and this reduces the number of garbage collections that must be invoked. Traditionally, almost all the numerical operations done in LISP are on these small integers.

Construction of Lists

Careful allocation of the address space of virtual memory alleviates only some of the problems of list processing in a paging environment. List structure, unless specifically and purposefully organized, tends to become random and thus defeats the advantages of the paging scheme. If only one-tenth of the existing list structure can reside in core, and it is referenced randomly, then nine out of ten data references, on the average, will require a drum operation.

An examination of list-handling processes indicates that lists are usually processed sequentially; that is, programs generally proceed down the elements of a list by taking the CDR of the successive tails. A process may be handling several lists at once, but will typically make numerous references to each. One of the best means of speeding up a paged LISP appeared to be linearizing lists and concentrating them on as few pages as

possible. To do this, a special CONS, or list-word constructing subroutine, was written. This attempts to assign a new list-word on a page already in use by the list of which this new word is a part. The algorithm is described below.

In constructing a new list pointer, a free word pair must be obtained from a free storage list. Instead of keeping one large free storage list as is done on the 7094 version of LISP, we have a separate free storage list on each page. Thus, the system can determine if a new pointer pair can be placed on a particular page. Using these free storage lists, we now construct a new pointer pair (a LISP dotted pair consisting of two 18 bit PDP-1 words) according to the following algorithm:

To construct Z = CONS [X;Y]:

1. If Y is not an atom and there is room on the page with Y then place Z on this page.

2. Otherwise, if X is not an atom and there is room on the page with X put Z on that page.

3. Otherwise, if there is a page in core with room place Z on this page.

4. Otherwise, place Z on a page in virtual memory with room. This page is found by searching an in-core table which gives the initial location of the free storage list on each page, if storage is available.

This algorithm tends to minimize cross linkages between pages and to limit any single structure to a very few pages. This has been born out by tracing through a number of fairly large

structures and computing the number of page crossings. When working with such a structure, it is unlikely that one will make references to more than a few pages for a relatively long period of time. Since these pages can reside in core, no drum references are needed. For example, in entering the definition of a function, the entire definition tends to appear on a single page. Thus, during the interpretation of a function multiple drum references are usually unnecessary.

When free storage is exhausted, garbage collection is necessary. A number of garbage collection schemes (4) have been invented and implemented in various versions of LISP. In some of these schemes, free storage is compacted by a folding process in which empty cells in lower storage are filled with the content of cells in higher storage. However, this is a very bad type of scheme for a paging environment, because this tends to effectively shuffle pointers, and make lists extend over more pages than are necessary. In our system, we simply mark used cells and collect storage on each page as it stands. In addition to this standard garbage collection algorithm, we also utilize another scheme for dumping onto secondary storage (magnetic tape) a compacted representation of the list structure in use in the system. This scheme (described in (4)) based on an algorithm first suggested by Marvin Minsky (5) has the desirable property that lists are, in general, linearized in the CDR direction.

SECTION IV

VARIABLE BINDINGS AND THE PUSHDOWN LIST

A number of schemes have been used in different versions of LISP for storing the values of variables. These include:

1. Storing values on an association list paired with the variable names.

2. Storing values on the property list of the atom which is the name of the variable.

3. Storing values in a special value cell associated with the atom name, putting old values on the pushdown list, and restoring these values when exiting from a function.

4. Storing values on the pushdown list.

The first three schemes all have the property that values are scattered throughout list structure space, and, in general, in a paging environment would require references to many pages to determine the value of a variable. This would be very undesirable in our system. In order to avoid this scattering, and possible excessive drum references, we utilize a variation on the fourth standard scheme, usually only used for transmitting values of arguments to compiled functions; that is,

we place these values on the pushdown list. But since we use an interpreter as well as a compiler, the variable names must be kept. The pushdown list thus contains pairs each consisting of a variable name and its value. The interpreter need only search down the pushdown list for the binding (value) of a variable.

One advantage of this scheme is that the current top of the pushdown stack is usually in core, and thus, drum references are rarely required. Free variables work automatically in a way similar to the association list scheme. In fact, the pushdown list is a guaranteed linear version of the association list.*

An additional advantage of this scheme is that it is completely compatible with compiled functions which pick up their arguments on the pushdown list from known positions, instead of doing a search. To keep complete compatibility, our compiled functions put the names of their arguments on the pushdown list, although they do not use them to reference variables. Thus, free variables can be used between compiled and interpreted functions with no special declarations necessary. The names on the pushdown list are also very useful in debugging, for they provide a complete symbolic backtrace in case of error. Thus, this technique, for a small extra overhead, minimizes drum references, provides symbolic debugging information, and allows completely free mixing of compiled and interpreted routines.

---

* With appropriate passing down of pushdown list pointers we can even achieve the same effect as the standard FUNARG device on the 7094 for functional arguments. This FUNARG device preserves local context well enough to handle the Knuth's "Man and Boy" compiler problem (Algol Bulletin 18.1).

SECTION V

PERFORMANCE

The proof of programming techniques is in the running. We have made a number of measurements on our system to test the validity of our assumptions. To test the hypothesis that our CONS algorithm was helping to minimize drum references, we ran a computation for about 35 minutes with a number of counters in various functions. The computation made was the compilation of approximately 10 pages of LISP functions which define an elementary programming language called GHOST. We were utilizing about 12,000 words of free storage; core can only hold about 3,500. Typical garbage collections reclaimed about 5,000 words. In this time we performed 31,000 CONS's, and 150,000 CAR's and CDR's (not counting those done in garbage collection). For these CONS's and CAR's-CDR's only 5,500 drum references were needed, or only about 2.5 per cent of the cases. If storage were distributed randomly, then the expected number of references should have ranged between 50 per cent and 70 per cent depending on whether the space in use for list structure was nearer 7,000 or 12,000 words. With the 2.5 per cent figure, we computed that the system spent 10 per cent of its time waiting for the drum. If the system had to go to the drum for 50 per cent of its data

references, it would be spending well over 90 per cent of its time waiting for the drum. Extracting the same information from a number of other runs for different users of the system indicates that the drum is referenced between 2 per cent and 10 per cent of the time. The types of jobs being done by the user were not recorded. The higher percentage was realized when the user was involved in many small on-line interactions with the system. However, it is in just this case that waiting for the drum is least costly to the user, since most of his time is spent thinking.

Another facet of the cost of using the drum should be mentioned. Even in problems in which all the storage needed can fit into core, you are paying a price for the drum facility. Instead of direct references to core, the system must go through a software map to determine core addresses. We compared the running time of one program on an in-core LISP system, and a paged LISP system, on an SDS-940. Despite the fact that all the data fit in core, this experiment indicated that we are losing a factor of about 2 in speed when we use the software map. Of course, if a hardware paging map were available this problem would vanish.

SECTION VI

SUMMARY

In summary, our LISP system surpassed our expectations and is proving a useful research tool for a number of artificial intelligence projects. Careful segmentation of system code, arrangement of data spaces by type, organization of list structure, and special attention to the binding of variables all contribute to the success of the system.

ACKNOWLEDGEMENTS

The authors would like to thank L. Peter Deutsch of Berkeley, California for writing (in LISP) the first version of the compiler used on this system, and for a number of stimulating discussions. The work reported here was supported by the Advanced Research Projects Agency, ARPA Order No. 627, Amendment No. 2, dated 9 March 1965.

REFERENCES

1. McCarthy, J. et al, LISP 1.5 Programmers Manual, M. I. T. Press, Cambridge, Massachusetts, 1964.

2. Berkeley, E. G. and D. G. Bobrow (eds.), The Programming Language LISP, Its Operation and Applications, M. I. T. Press, Cambridge, Massachusetts, 1966.

3. Corbato, F., E. Glaser et al, "The MULTICS System," Proc. FJCC, Spartan Press, Baltimore, Maryland, 1965.

4. Bobrow, D. G., "Storage Management in LISP," Proc. IFIP Conf. on Symbol Manipulation Languages, North Holland. (In preparation)

5. Minsky, M. L., "A LISP Garbage Collector Algorithm Using Serial Secondary Storage (rev.)," Memo 58, Artificial Intelligence Group, M. I. T., Cambridge, Massachusetts, 1963.

6. Cohen, J., "A Use of Fast and Slow Memories in List Processing Languages," Comm. ACM, January 1967.

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, Incorporated
50 Moulton Street
Cambridge, Massachusetts

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE

The Structure of a LISP System Using Two-Level Storage

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

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

Bobrow, Daniel G.
Murphy, Daniel L.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
4 November 1966	20	6

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
AF19(628)-5065 ARPA Order No. 627, Amend. 2	Scientific Report No. 6
b. PROJECT XXXXXXX No. 8668	
c. DOD ELEMENT 6154501R	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d. DOD SUBELEMENT n/a	AFCRL-66-774

10. AVAILABILITY/LIMITATION NOTICES

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
Hq. AFCRL, OAR(CRB) United States Air Force L. G. Hanscom Fld., Bedford, Mass.	Advanced Research Projects Agency

13. ABSTRACT

In an ideal list-processing system there would be enough core memory to contain all the data and programs. This paper describes a number of techniques used to build a LISP system which utilizes a drum for its principal storage medium, with a surprisingly low time-penalty for use of this slow storage device. The techniques include careful segmentation of system programs, allocation of virtual memory to allow address arithmetic for type determination, and a special algorithm for building reasonably linearized lists. A scheme is described for binding variables which is good in this environment and allows for complete compatibility between compiled and interpreted programs with no special declarations.

DD FORM 1 JAN 64 1473

Unclassified
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
List Processing						
LISP						
Paging Systems						
Dynamic Storage Allocation						
Two Level Storage Allocation						

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.