THE ENTROPY OF ADDRESS FIELDS IN PASCAL PROGRAMS

by

James Jatczynski

Memorandum No. ERL-M601

September 1976

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

Processors offering very compact program and data representation can make the support of extremely large applications economically feasible because of their relatively smaller memory requirements. Here we examine the address fields of several PASCAL programs to determine a lower bound on the effectiveness of compact address field encoding. We find reductions in program size ranging from 6 to 12 per cent under the most efficient encodings.

The Entropy of Address Fields in PASCAL Programs (1)

1 Summary

Considerable effort has been invested in tailoring processor hardware to specific programming languages [1, 2, 3, 8, 9]. In particular, recent interest in LISP-language [see 6] processors has embraced the goals of economical provision of sufficient processing capacity to support very large applications [1, 2, 3]. This interest is prompted largely by the needs of the artificial intelligence community, but many disciplines could benefit from inexpensive large-scale processors. The requirement of high capacity implies a need for fast hardware supporting a large address space whereas the need for economy implies extremely efficient hardware architecture.

Since primary memory contributes significantly to processor cost and size, architectural techniques which allow compact program and data coding offer a way to limit processor cost. Some work on compact representations has been done, particularly for the LISP language [3, 4, 6].

Here we examine the address fields in PASCAL programs in order to identify addressing characteristics which imply effective means of program (particularly address field) compaction. Our main goal is to determine a lower bound for the size of compact address field encodings. CDC 6400 PASCAL was chosen because

----
(1) This work was supported in part by Office of Naval Research Contract N00014-76-C-0013.

of the expectation that its addressing behavior approximates that of many other languages (including compiled LISP) and the availability of a variety of PASCAL programs.

We find here that the large address field size and addressing architecture of many machines is not justified by empirically observed addressing characteristics. For example, in a compiler requiring 20084 words of storage and containing 16963 addressing instances, the address field entropy was only 9.02 bits.

Thus, it is clear that efficient program storage utilization requires hardware that more closely mirrors the structure of the languages that are used on the machine.

2 Introduction

2.1 Background

Many findings and proposals related to compact data and program representation are found in the papers of Clark and Green [4] and Deutsch [3].

Clark and Green examine the list structure data present at the end of execution of five large LISP programs. Their measurements reveal considerable regularity in the usage of pointers, particularly pointers to lists as opposed to pointers to atoms. For example, about 70% of all CDRs point to lists and 25% are NIL. More significantly, of all pointers to lists, over 80% point to storage locations less than 128 cells distant from the cell containing the pointer; in addition, they find that certain list linearization techniques improve this locality of reference. They state that "the regularity of LISP list structures . . . suggests that compact encodings of pointers might greatly reduce the bit requirements of a list cell," and go on to determine a lower bound on the efficiency of these encodings for the given programs by computing the average information content or entropy of the pointer fields. The maximum CAR plus CDR entropy they find is 15.48 bits--this when 36 bits are actually used to hold these pointers in the given implementation. A clear case for the desirability of compact data representation in LISP is made, and several practical techniques are presented, particularly the hash link scheme of Bobrow [7].

Deutsch outlines the design of a LISP machine with compact representation of compiled LISP programs. The techniques he presents are based on the structure of the LISP language (and many other high level languages) and apparently are sound although he presents no a priori theory or measurements of LISP programs which would suggest them. His main ideas are 1) to use a local and a global name table to control all data accessing and thereby allow address fields (which then refer indirectly to data through these tables) to be short and 2) to closely mirror the language structure in the hardware. Deutsch claims that programs for his machine are consistently one-third to one-fourth the size of equivalent INTER-LISP compiled programs. Deutsch's design does not take complete advantage of knowledge of the language structure; even greater compaction could be achieved by taking this structure into account.

The combination of these findings lead us to believe that compaction of LISP programs and data by a factor of at least two over other "good" implementations is feasible. It is also clear that program compaction is possible for most languages although data compaction potential is not as clear for languages with pointer data due to the irregularity of their data structures which use pointers when compared to those of LISP.

2.2 The Entropy Concept

From now on we will be concerned only with the collections of all address fields found in the code produced by the PASCAL 6000-3.4 compiler for particular programs. Our goal is to obtain a lower bound on the size of all possible address field encodings for each given program. This information can allow us to judge the effectiveness of compaction techniques that we may consider and can also point out possible compaction techniques. However, these topics will only be briefly discussed in this paper.

Consider each address field as a message drawn from the set of all possible address fields

$$A = \{a_1, a_2, \ldots, a_n\}$$

Let

$$F = \langle f_1, f_2, \ldots, f_m \rangle$$

be a list of all address fields which occur in a given program. Clearly all members of F are drawn from A, and members of F need not be distinct. With $|A|$ the number of elements in A and $|F|$ the number of elements in sequence F, let

$$p_i = n_i/|F| , \qquad 1 \leq i \leq |A|$$

where n_i is the number of occurrences of a_i in F.

We now define the average information content or entropy of messages in the sequence F as

$$H(F) = -\sum_{i=1}^{n} p_i \log_2 p_i .$$

Thus, for example, the entropy of a set of eight equally probable messages is 8.

For a given sequence F, $R_{min} = H(F)^{\bullet}|F|$ gives the minimal number of bits required to encode the sequence. An important point should be noted: R_{min} is the minimal number of bits needed to unambiguously encode exactly the set of addresses F found in the given program; architectural alterations which affect the set of addresses F obviously affect R_{min}. Here we consider only changes in address encoding, not addressing architecture. Also, to achieve the minimal encoding, the processor would, in a sense, have to understand the meaning of each encoded address (i.e. be able to translate it into a machine address). This is possible, but the overhead needed to achieve it clearly precludes the practical attainment of the minimal encoding.

2.3 Scope and Limitations

We examine a small number of programs although there is a considerable variety in program size. The primary intent here is to describe the observational results and very briefly consider the determinants of these results. Details about methods of program compaction will not be discussed; some such discussion is found in [1, 2, 3].

Since the object programs we examine are all produced by the same compiler, it is not clear that the results are universal. However, it is believed that similar results would be found for a variety of languages, particularly procedure oriented algebraic languages.

3 Discussion

3.1 Experiments

A program was written which accepts as input load modules produced by a CDC 6400 loader from object modules produced by the PASCAL 6000-3.4 compiler. It locates all address fields in the given program text and computes certain relevant statistics. For some purposes, the set of address fields is divided into three classes--jump targets, constants, and load/store (data) addresses. In other cases, the address set is considered as a whole.

Overall and for each of the three address classes a count is kept of the number of address fields found. Also, the number of occurrences of each specific address value is recorded. These figures allow us to compute entropy values for each address class and for the set of all addresses taken together.

Jump addresses are handled in a slightly different way than constant and data address fields. The latter are taken as is to form the message sequence F. However, jump addresses are first converted to program counter relative values before being added to F. This is done because relative addressing appears to be an attractive way of keeping jump addresses small, and it is therefore more interesting to know the address set entropy under the relative addressing condition. In a sense, this is also done implicitly for data addresses most of which are offsets relative to an index register which serves as the runtime stack pointer.

Only addresses in the PASCAL produced code are analyzed. Addresses within system routines are not examined.

3.2 Programs Analyzed

Six PASCAL-produced load modules were examined. These are listed in Table 1 along with their sizes in 60-bit words. The primary reason for the choice of this set of programs was the wide range of program size. Programs 1 and 2 differ from the rest slightly in that they use few system routines while the others use system routines heavily and therefore have many jump ad-

Table 1
List of Analyzed Programs

Name (and Abbreviation)	Size (words)
1 PASCAL 6000-3.4 Compiler (PASCAL)	20084
2 PASCAL-S Translator/Interpreter (PASCALS)	11800
3 Basic Minicomputer Assembler (ASSM)	4224
4 Wirewrap Aid (WIRE)	2624
5 Hashing Technique Demonstration (HASH)	1728
6 Tower of Hanoi Solver (TOWER)	164

dresses directed toward these routines.

4 Results

Table 2 gives the number of occurrences of each type of address field in each program. Also given is the percentage of address occurrences of each type. The percentage of addresses falling into each category is seen to be relatively constant over programs.

Table 3 gives the entropy values corresponding to the sets represented by each entry in Table 2. Many entropy values are quite small compared to the corresponding number of address occurrences. For example, for the 16963 address fields in PASCAL, the entropy of 9.02 accounts for only 16963*9.02 = 153006.26 bits whereas the CDC 6400 actually uses 16963*18 = 305334 bits or approximately twice the minimal bit requirement to represent the address fields. This computation is performed for all programs and address classes and appears in Table 4.

Figures 1a-1e show the distribution of occurrences of relative jump addresses for each program (except TOWER). A logarithmic scale is used for jump distances with a minus sign appended to the logarithms of negative relative addresses. These data elucidate address clustering and the potential for program compaction via long and short instructions.

Table 2
Address Field Occurrences in Analyzed Programs

Program	Number of Address Occurrences and (per cent)			
	Jump Addresses	Constants	Data Addresses	All Addresses
PASCAL	5505 (33)	5139 (30)	6319 (37)	16963 (100)
PASCALS	1808 (35)	1426 (27)	2014 (38)	5248 (100)
ASSM	911 (31)	726 (25)	1305 (44)	2942 (100)
WIRE	335 (23)	453 (32)	652 (45)	1440 (100)
HASH	162 (29)	137 (24)	268 (47)	567 (100)
TOWER	21 (27)	21 (27)	35 (46)	77 (100)

Table 3
Entropy Values of Address Sets

Program	Address Set Entropy			
	Jump Addresses	Constants	Data Addresses	All Addresses
PASCAL	9.61	6.13	8.67	9.02
PASCALS	8.71	5.35	7.84	8.39
ASSM	8.68	5.58	7.32	8.31
WIRE	7.87	4.05	6.31	6.92
HASH	7.05	2.87	6.10	6.69
TOWER	4.20	2.89	4.38	5.12

Table 4
Minimal vs. Actual Bit Requirements for Address Fields

Program	Minimal bit requirements divided by actual usage on the CDC 6400				Per cent reduction in program size using minimal encoding
	Jump Addresses	Constants	Data Addresses	All Addresses	
PASCAL	.53	.34	.48	.50	13
PASCALS	.48	.30	.44	.47	7
ASSM	.48	.31	.41	.46	11
WIRE	.44	.23	.35	.38	10
HASH	.39	.16	.34	.37	6
TOWER	.23	.16	.24	.28	10

Figure 1a. Jump Address Distribution of PASCAL

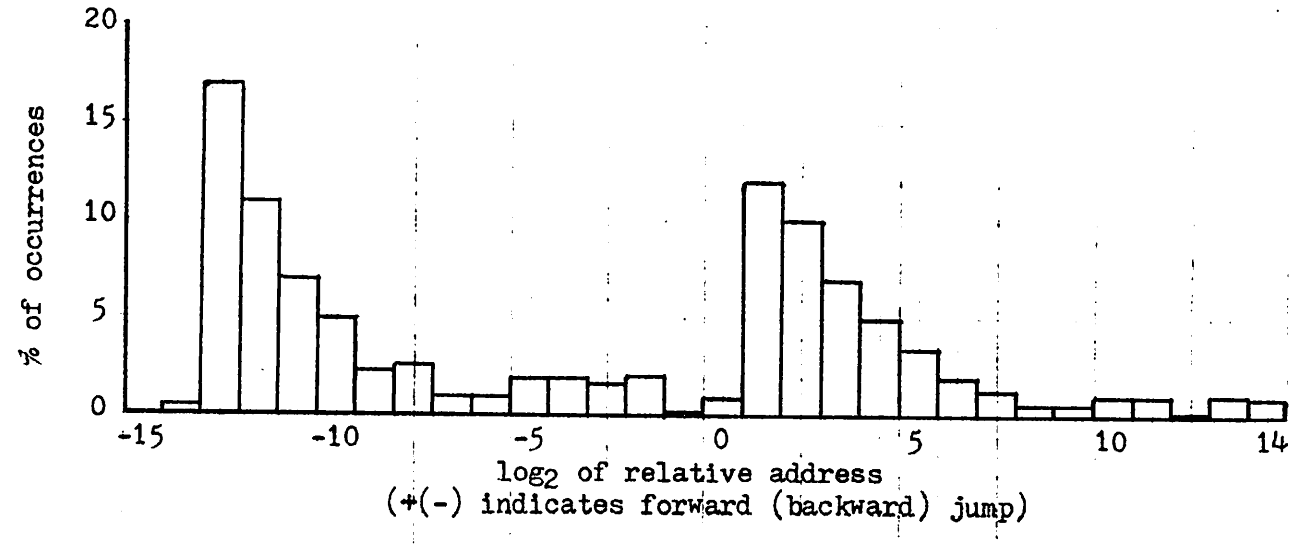


Figure 1b. Jump Address Distribution of PASCALS

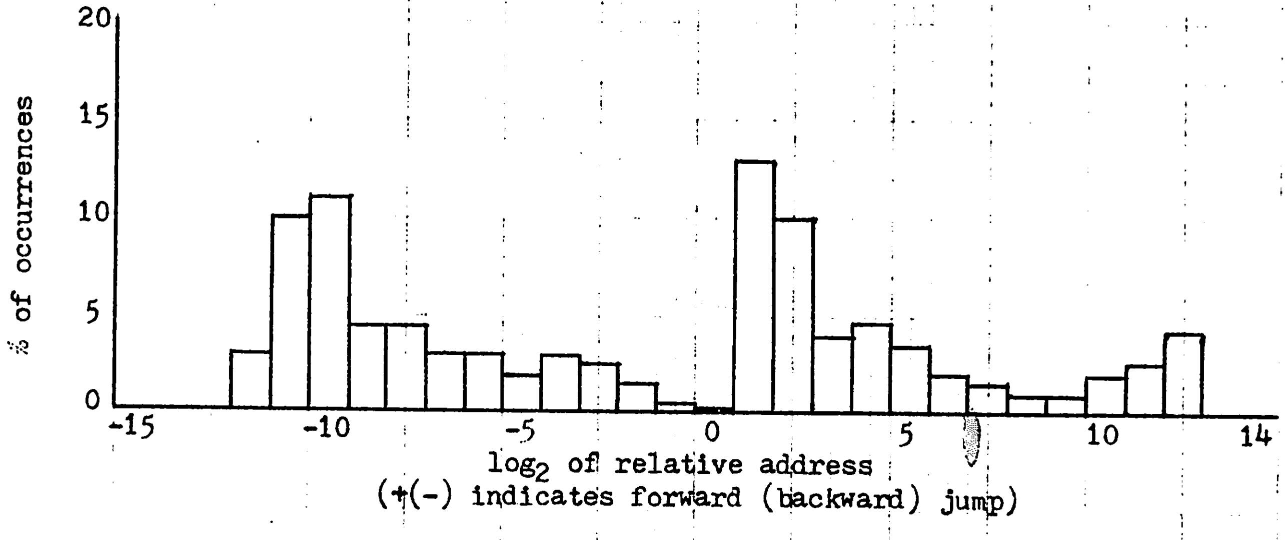


Figure 1c. Jump Address Distribution of ASSM

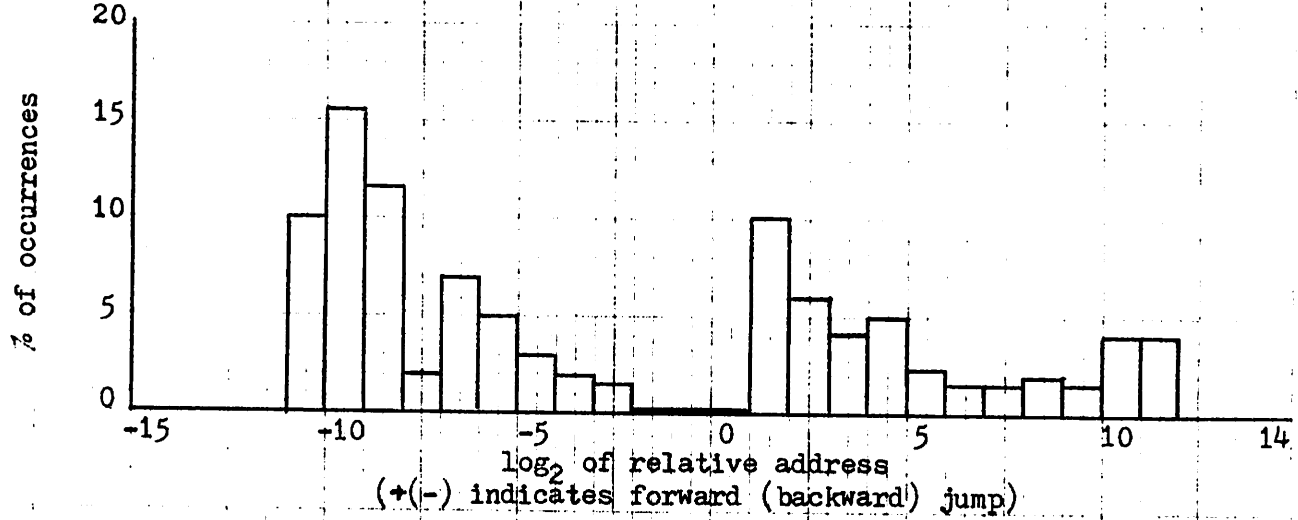


Figure 1d. Jump Address Distribution of WIRE

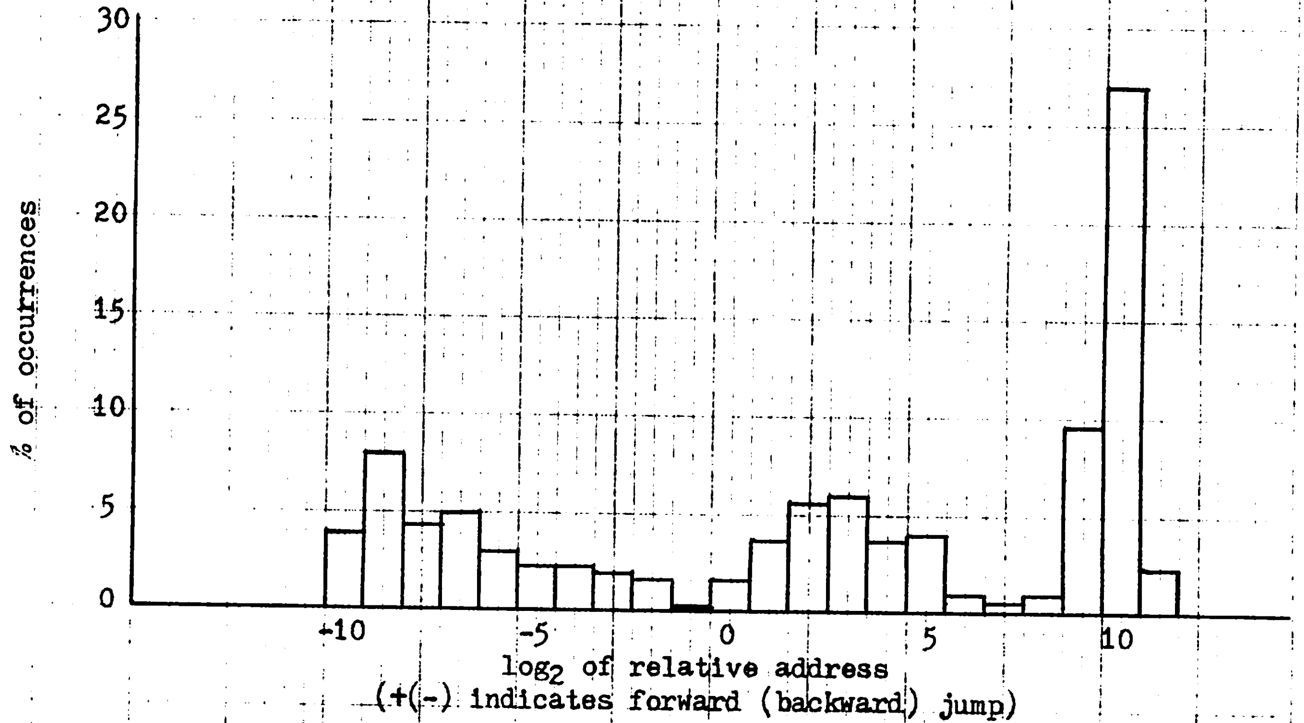
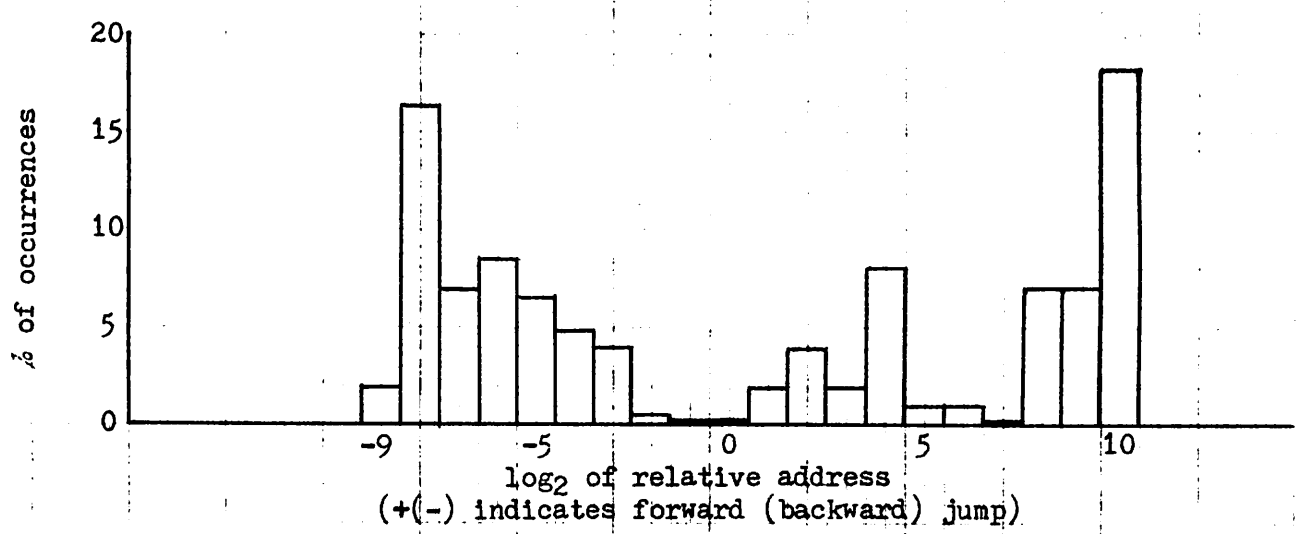


Figure 1e. Jump Address Distribution of HASH



5 Conclusions and Recommendations

Several important implications are noted here.

First a clear case is made for the potential effectiveness of address field compaction in certain programs. For example in PASCAL address fields account for 25% of the program's storage requirement. Minimal encoding of address fields would reduce the total program size by about 12%, and simple practical techniques promise at least half this reduction. For example, consider Figure 1a which shows that 48% of the jump targets in PASCAL are within 128 cells of the jump instruction. The use of a short instruction (8-bit address field) in each of these cases would reduce program size by about 2.5% (the CDC 6400 has no such short instructions). Similar short instructions for constant and data references would provide comparable reductions. Unfortunately, for the program HASH, a minimal address field encoding would reduce program size by only 6%. Thus it is also clear that compaction of address fields alone may not significantly reduce program size and that other organizational techniques relating to addressing may be necessary. Table 4 shows the reduction in program size which would result from a minimal address field encoding.

Second, the observation that entropy values are small with respect to the number of addressing occurrences indicates considerable duplication in the occurrence of address values. For example, in PASCAL there are 16963 address fields; but, whereas

$\log_2 16963 = 14.05$, the entropy value is only 9.02. In fact, the number of distinct addresses is about half the total number of address fields. This suggests that reference to a table of address values by using small offset fields in instructions may offer a means of address field compaction. Under reasonable assumptions it is easy to see that such a table of addresses along with small offset fields in instructions takes no more space than the original CDC 6400 implementation; but with judicious ordering of addresses in the table and concomitant choice of short instructions where possible, the potential for compaction is considerable. It is also apparent from data not given here explicitly that the use of global and local address tables similar to those proposed by Deutsch would be an effective address compaction tool.

Third, the distributions of relative jump addresses seen in Figures 1a-1e indicate that variable length address fields may be useful in achieving compaction. For the given programs, between 36% and 48% of all jump targets are within 127 cells of the jump instruction itself. Thus up to 48% of all jumps could be handled by an 8-bit address field.

Acknowledgement. I would like to thank R. J. Fateman for his helpful criticism.

6 References

[1] R. Greenblatt, The LISP machine, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Working paper 79, November 1974.

[2] Tom Knight, CONS, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Working paper 80, November 1974.

[3] L. P. Deutsch, A LISP machine with very compact programs, Third International Joint Conference on Artificial Intelligence, Stanford, California, 1973, pp. 697-703.

[4] D. W. Clark and C. C. Green, An empirical study of list structure in LISP, Xerox Palo Alto Research Center working paper, December 1974, To appear in Comm. ACM.

[5] J. Moses, The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem, ACM SIGSAM Bulletin No. 15 (July 1970), pp. 13-27.

[6] J. Allen, The Anatomy of LISP, To be published by McGraw-Hill.

[7] D. G. Bobrow, A note on hash linking, Comm. ACM 18, 7 (July 1975), 413-415.

[8] H. Weber, A microprogrammed implementation of EULER on IBM System/360 Model 30, Comm. ACM 10, 9 (Sept. 1967), pp. 549-558.

[9] W. Lonergan and P. King, Design of the B 5000 system, Datamation 7, 5 (May 1961), pp. 28-32.