01 Jan 1967

Reviews Of Books And Papers In The Computer Field

James H. Tracey
Missouri University of Science and Technology

Recommended Citation

J. H. Tracey, "Reviews Of Books And Papers In The Computer Field," *IEEE Transactions on Electronic Computers*, vol. EC thru 16, no. 2, pp. 234 - 242, Institute of Electrical and Electronics Engineers, Jan 1967. The definitive version is available at https://doi.org/10.1109/PGEC.1967.264588

Reviews of Books and Papers in the Computer Field

DONALD L. EPLEY, REVIEWS EDITOR
D. W. FIFE, A. J. NICHOLS, A. I. RUBIN, H. S. STONE
ASSISTANT REVIEWS EDITORS

Please address your comments and suggestions to the Reviews Editor: Donald L. Epley, Department of Electrical Engineering, University of Iowa, Iowa City, Iowa 52240.

A. SWITCHING AND AUTOMATA THEORY

R67-17 Logic, Automata, Algorithms—M. A. Aizermann, L. A. Gusev, L. I. Rosenaur, I. M. Smirnov, and A. A. Tal (USSR: Government Publications of Physical Mathematical Literature, 556 pages).

This book covers some of the most diverse subject matter I have seen in the area of automata theory–sequential machines. While treading some of the same waters as Ginsburg and Gill, the book also treats relay circuits, Gödels theorem, Church's thesis, and some Turing machine theory. By virtue of length and close packing the book is able to range through this volume of material, without being superficial in its treatment.

There is, of course, some switching theory of the sort one finds in McCluskey and Harrison, *sans* the extensive combinational network design techniques. The book concentrates more on the Huffman and Moore treatment of finite-state machines and realizations and minimization of number of states, then examines what can be computed, algorithms, experiments somewhat *à la* Moore, etc.

All in all this seems an excellent book of a type we have not yet generated. The material is carefully and clearly presented. The choice of topics, notation, figures and general presentation are distinctly Soviet, and I suspect this book could be quite popular abroad. Unfortunately, there seems to be no English translation available.

THOMAS C. BARTEE
Harvard University
Cambridge, Mass.

R67-18 Realization of Input-Output Relations by Sequential Machines—A. Gill (*J. ACM*, vol. 13, pp. 33–42, January 1966).

This paper treats an important problem in sequential machine theory, the construction of sequential machines from input–output specifications. The input–output specifications consist of a tabular arrangement of input–output sequences, where each sequence is finite in length. The author determines necessary and sufficient conditions for the set of input–output sequences to be realizable by a sequential machine, and then formulates an algorithm for the construction of such a machine when the conditions are met. Sequential machines considered here are restricted to finite, deterministic, synchronous, completely specified, Mealy-type, sequential machines. An important consideration is that no initial state is stipulated in the contemplated machine. In other words, every state may serve as an initial state. As a result, one cannot uniquely associate an output sequence with each input sequence.

The author proves that a set of input–output sequences (each of finite length) is realizable if and only if it is compatible. This compatibility property is precisely defined and may be tested for. Realization algorithms are presented for those sets of input–output sequences that satisfy the compatibility property. As one would expect, it can become quite tedious to generate a state diagram from a large set of compatible input–output sequences. In the process, quite a number of redundant states are introduced; but standard simplification techniques may be used to simplify the machine. It is fortunate at this point that the generated machine is completely specified so that machine simplification is relatively easy to carry out.

The testing of input–output sequences for realizability and the synthesis of even relatively simple sequential machines can become quite lengthy with the techniques of this paper. Nevertheless, the author has provided valuable insight into the difficult problem of translating machine specifications into a sequential machine model.

J. H. TRACEY
University of Missouri
Rolla, Mo.

Editor's Note: Due to a change in editors, two reviews of the following book were inadvertently solicited. Since the following review adds materially to the earlier review (R66-42, August 1966), it was included in this issue.

R67-19 Introduction to the Theory of Switching Circuits—E. J. McCluskey (New York: Electrical and Electronic Engineering Series, McGraw-Hill, 1965, 318 pages.)

When an expert in a particular field sets about to write a book on his pet subject, he is always tempted to tell the reader all that he knows. To resist this tendency and to keep a proper balance between completeness and oversize requires courage, a perfect knowledge of the subject, and a strong didactic sense. Prof. McCluskey's latest book provides us with the proof that this ideal can indeed be accomplished.

The word "introduction" in the title of the book should not be misinterpreted. It does not mean that the book represents an elementary treatment of the theory of switching circuits, but rather that the careful reader will really be well introduced into the field of switching theory and will thereafter be able to understand most of the current literature on the subject. Furthermore, to quote the author's words, "a vigorous attempt has been made to include only topics that will be of lasting general importance. In keeping with this objective, . . . the emphasis of this book is on algorithms and general theory." After reading the book, one is convinced that the author has been true to his word and that his work is not likely to be outdated before many years. However the material included in the book has

been selected on the basis of its practical importance for the engineer rather than its theoretical interest for the mathematician.

The basic switching components are discussed in Chapter 1, including contacts, diodes, transistors, magnetic elements and cryotrons. Chapter 2 deals with number systems and codes. Boolean algebra is introduced in Chapter 3, first as a model for switching circuits, later in the chapter as an abstract mathematical system.

The methods for the simplification of logical functions are discussed in Chapter 4, with the emphasis on two-stage logic. It is rather a pity the map method has not been extended as it should for more than four variables, i.e., in the form of a single map with its rows and columns numbered in the reflected binary code. If the numbering is indicated graphically, in the form of a digital measuring scale, the optimal grouping of the ones is readily achieved up to 8 variables, owing to the adjacency properties of the reflected binary system. The method of iterative consensus is carefully expounded, but there again it is a pity Tison's algorithm, forming the consensus terms in succession with respect to the different variables, is not mentioned. There is a clear discussion of the author's numerical method, which is made even more straightforward by the use of the octal number system. Multiple-output circuits are treated by the three methods —the map method, the Quine-McCluskey method, and the iterative consensus method.

The last three chapters, devoted to sequential switching circuits, probably form the most interesting part of the book. The integrated treatment of both pulse-operated and level-operated sequential circuits, including clocked systems, is particularly noteworthy. Chapter 5 deals with the analysis of sequential circuits of both types by such methods as the flow-table and the state diagram. Chapter 6 presents synthesis techniques, including the determination of maximum compatibility sets for the reduction of incompletely specified flow tables. Finally the dynamic behavior of sequential switching circuits, taking into account the consequences of spurious delays, is considered in Chapter 7. Static, dynamic, and essential hazards are discussed in detail.

Each chapter terminates with a large number of interesting problems, many of which are used for treating specific applications or for introducing some of the more specialised concepts not dealt with in the body of the book.

Summing up, this book is a most welcome addition to the basic literature on switching theory.

P. NASLIN
French Corps of Armaments
Paris, France

B. READ-ONLY STORES

R67-20 Design of a Printed Card Capacitor Read-Only Store—J. W. Haskell (*IBM J. R&D*, vol. 10, pp. 142–157, March 1966).

This paper contains a well-written and detailed description of a read-only store used in a current machine. It includes discussion of the engineering of the memory and presents much information on both the physical construction and the electrical operation of the system. Included also are numerous worst-case calculations and associated curves.

I recommend this paper, particularly to those in our business who, after developing the first glimmer of an idea for a new memory (and possibly fabricating an experimental four-bit array), discount the efforts involved to develop it further as "mere engineering details." While no cost figures are given in the paper, I believe it supports my contention that terms such as "*potentially* very low cost" are also to be taken with a grain of salt.

MORTON H. LEWIN
RCA Laboratories
Princeton, N. J.

C. ANALYSIS OF COMPUTER SYSTEMS

R67-21 Time-Shared Computer Operations with both Interarrival and Service Times Exponential—B. Krishnamoorthi and Roger C. Wood (*J. ACM*, vol. 13, pp. 317–338, July 1966).

Along with the increasing importance of the technique of time-sharing, there has been a marked growth in interest in analyzing and predicting queuing behavior in computer systems, and in optimally controlling the queuing disciplines by which a limited computing resource is allocated among waiting tasks. The whole related "utility" concept of computers depends upon sensible handling of the difficult problems arising from intermeshing large numbers of randomly occurring demands for computing service. The early history of the telephone points very plainly in this direction, and has very obvious parallels to the computer utility.

This paper represents a contribution to this rising new technical area. It explores a queuing theoretic model for a system consisting of a single time-shared central processing unit with a fixed number of active users. Its objective is to analytically determine probabilistic measures of performance which should aid in the rational selection of the quantum interval.

The model makes use of assumptions that interarrival times and total service intervals are simple exponentially distributed random variables, and it describes two different round-robin queuing strategies for assignment of fixed slices (quanta) of time to waiting jobs. Swapping time is modeled by "loading" the execution time, but other overhead influences are neglected. As a consequence of the assumptions, the model is shown to be susceptible to treatment by the time-honored method of embedded chains. Calculations are made of 1) an approximation to the limiting distribution of the number of users waiting, 2) mean and variance of cycle time (under two different definitions of "cycle time"), and 3) mean time in system conditional on execution time.

The model is one more in a number of queuing models for time-shared and multiple-access computing systems which have been proposed and analyzed.[1–5] In some of these cases results have been obtained analytically (as here), in others numerically,[6] and in still others by simulation. Each of these models has given some general insights into the control and design of such computer systems, and as such have represented significant contributions to the time-sharing art. Although the insights to the time-shared system design developed by the authors of this paper are fairly limited and specific, the generalizations attempted are provocative and interesting. Further exploration of their results, plotted over other ranges of system parameters than those presented in the paper, should yield still more general insights and are worthy of additional study.

This paper is an excellent example of the analytical solution of a queuing model for a computer system by the valued methods of embedded chains and moment generating functions. The mathematics is clearly executed and well explained. Thus, the paper should encourage further exploitation of these and related techniques in the analysis of modern computer systems.

VICTOR L. WALLACE
University of Michigan
Ann Arbor, Mich.

[1] D. W. Fife and R. S. Rosenberg, "Queueing in a memory-shared computer," *Proc. 19th Nat'l Conf. ACM*, Philadelphia, 1964.
[2] A. L. Scherr, "An analysis of time-shared computer systems," M.I.T., Cambridge, Mass., Project MAC-TR-18, June 1965.
[3] E. G. Coffmann, "Stochastic models of multiple and time-shared computer operations," Dept. of Engrg., University of California, Los Angeles, Report 66-38, June 1966.
[4] J. L. Smith, "An analysis of time-sharing computer systems using Markov models," *1966 Spring Joint Computer Conf., AFIPS Proc.*, vol. 28, pp. 87–95.
[5] D. W. Fife, "The optimal control of queues, with applications to computer systems," Cooley Electronics Laboratory, University of Michigan, Ann Arbor, Technical Report 170, October 1965.
[6] V. L. Wallace and R. S. Rosenberg, "Markovian models and numerical analysis of computer system behavior," *1966 Spring Joint Computer Conf., AFIPS Proc.*, vol. 28, pp. 141–148.

D. PROGRAMMING

R67-22 The LISP 2 Programming Language and System—P. W. Abrahams and C. Weissman (*1966 Fall Joint Computer Conference, AFIPS Proc.*, vol. 29, pp. 661–676).

LISP 2 will one day stand on its own legs and be evaluated on its own merits. But in its first public appearance it is in the role of a child of famous parents. Comparisons are invited when a child is brought into the world to carry on and eventually take over his parents' business. Of course, people get old and eventually retire, if for no reason other than that they can no longer do what they were once able to do. Computer languages do not degenerate in the same way. When they are deliberately replaced it must be that the new generation addresses itself to problems that are *effectively* outside the scope of the old one. This point is taken up again below.

LISP holds an honored place alongside FORTRAN, ALGOL, and IPL-V as an important tool and, what is more important in the long run, as a seminal construct. That the culture spawned by FORTRAN and ALGOL is ubiquitous is obvious and need not be further discussed here. The number of people fluent in IPL-V is very small. The importance of that language in historical context is that it explicitly dealt with and exploited dynamically changing data and program *structures*. It planted the list-processing seed. At least traces, and usually much more, of its issue can be found in every modern product of the computer world. The LISP culture also has few active participants. However, it is immensely vigorous. One sign of this vigor is that new LISP interpreters and compilers appear with considerable regularity. Another is that extremely difficult problems (e.g., symbolic integration, visual object recognition, programmed proof procedures, mathematical assistants) continue to be cast in LISP programs. But the spermatic character of the LISP idea reveals itself in that it has permeated research in the foundations of computation. LISP *is* a topic in recursive foundation theory. It is *built on* the lambda calculus. Mathematical theorems *about* and *in* LISP are not ad hoc or a posteriori constructs clothing things not basically logically structured in mathematical costume.

What is the beauty and from whence derives the power of LISP? The beauty of LISP is that it is so utterly *simple*. The first few pages of the LISP 1.5 manual tell all that really needs to be told. All the rest is elaboration and example. The word for that is "elegance." The power of LISP is a function of its elegance. LISP is *not* a list processor. To be sure, its basic internal data type is the list structure. Indeed, the programmer may manipulate list structures explicitly. But large programs are written in which explicit list manipulation is not the major issue. The essence of LISP is that it is a *functional* language. And by this is meant that it applies functions to arguments. At this stage of the development of computation it almost need not be said that the latter may again be functions applied to arguments. It is, however, altogether uncommon that functions may themselves be applicative expressions, that, in other words, a program may execute a process (evaluate an expression) the result of which is a function (not a function name) which may then (or later) be applied to arguments. That unlimited nesting of such expressions within expressions is permitted goes without saying. Similarly for the fact that functions may be recursive in complicated ways. However, another word should be said about recursion. This is that the basic list organization of LISP and the provision of the operators CAR and CDR which yield the first element of a list and a list with its first element removed, respectively, makes recursive operations on complex data structures quite easy and, more importantly, natural.

Another strength of LISP must be mentioned because of what follows: It is possible—even normal—to have a program *construct* an expression and then have that expression evaluated. In other words, data can be treated as program. It is, of course, also true that a LISP program may operate on itself, i.e., that program may be treated as data.

Granted all the beauty, elegance and power described here, what then motivated anyone to create a child to replace this famous person? That LISP 2 is so intended follows from its name. Does the new package repair certain faults of the old one or does it take advantage of new opportunities, perhaps new ideas, to enhance and expand it?

The authors criticize LISP 1.5 (the current form of LISP) for not having "a convenient input language" and for treating purely arithmetic operations inefficiently. The first of these criticisms is met by providing a preprocessor to convert very ALGOL-like source level programs to very LISP 1.5-like internal language programs. The internal language is an intermediate stage. It is finally compiled into machine code. The arithmetic inefficiency difficulty is met by introducing type declarations. These give the compiler sufficient information *about* the program being compiled to permit generation of efficient code.

Here, by the way, is the first public admission on the part of what has sometimes been called the LISP "priesthood" that there is anything "inconvenient" about LISP notation. Indeed, the heavily parenthesised LISP notation is sometimes seen as a blessing (even if in disguise) in that it keeps the priesthood small—the price of admission, i.e., having to learn to balance endless series of parentheses, being too high for most ordinary humans. The removal (or at least reduction) of artificial tariffs can be greeted only with pleasure.

The efficiency issue is more pervasive than appears at first glance. Behind its acknowledgement lies an even more important recognition—namely, that many important problems demand information processing facilities of disparate kinds, i.e., that "symbol manipulation" is not the only tool required by the serious workman. It is here that the word "effective" as used above becomes operative. For no one argues that LISP (or, for that matter, FORTRAN) is not "universal" in that programs prescribing the computation of any computable number cannot be written in it. But there is a difference between *practical* computability and computability. (This difference is, by the way, becoming every more important as the maximum delays people and things can be made to endure are more and more determined by real-time criteria.)

It is in the service of the newly valued catholicity that the major departures from classical LISP have been ordained. The starting point is that efficient arithmetic is desired. Type declarations permit compilation time identifier discrimination and thus aid in generation of code appropriate to each task, hence efficient code. But what kind of arithmetic? Well, integer, floating-point, Boolean, multiple precision, ... But the storage of multiple precision numbers raises difficult problems in dynamic storage management, precisely the same ones arising out of attempts to allocate and recover array and matrix space.

The problem is that programs will demand blocks of *contiguously addressable* storage at essentially unpredictable times. If fewer cells are demanded than are on the free storage list, but no *contiguous* block of the required size is free, then the program has run out of space. But, except for poor organization, space is actually available. That situation cannot be allowed to persist. The solution adopted by LISP 2 is a so-called "compacting garbage collector." As in classical LISP, abandoned data structures are permitted to accumulate as long as there is no shortage of free space. They are not collected until a new pool of available space *must* be formed. In the classical LISP garbage-collection cycle, abandoned cells are marked as being again available by being strung into one monolithic free storage list. The process is roughly analogous to stringing a thread through all newly available cells. In a sense, the thread moves but the cells stay in place. In compacting garbage collection, on the other hand, the number of cells so freed is counted and the data contained in that number of cells starting from the top of the original list of available space are moved into the freed cells. All appropriate pointers in all data structures are modified accordingly. The new free storage list then consists of the largest possible block of contiguously addressable cells. Clearly, this latter scheme may involve the transfer of considerable amounts of information, hence, a proportionate expenditure of processing time.

One important advantage of dynamic storage allocation is that space is used over and over again, thus lowering the minimum storage requirements of a given program over what it would have had to have been under a rigid, static storage preassignment scheme, e.g., one governed entirely by DIMENSION statements. Nevertheless, there is some such minimum associated with any given program and the data structures on which it is to work. It is precisely in the complex problem areas for which LISP was designed that reliable estimates of such minima are very hard to make. The original list of available space is, therefore, made as large as possible—in batch processing usually the remainder of core after all essentials have been loaded. But the larger that list, the longer the garbage-collection cycle.

Recall that garbage collection is not initiated until the free storage list cannot satisfy a demand for free space—whether that demand be for a single cell or for a block of cells. But it is in the nature of the scheme here outlined that all substantive computation must cease while the list of available space is reorganized. The time required to collect garbage on a typical 7094 LISP 1.5 system is on the order of 0.5 second. That is for a machine with a 32 000-word memory of which about 10 000 words are devoted to available space. LISP 2 is meant for machines with possibly 256 000 words of core storage of which 200 000 words might well be allocated to the free list. Compacting probably doubles garbage-collection time. Thus, a garbage-collection cycle in a LISP 2 system running on a 256 000 machine of 7094 speed might take 20 seconds! Such lengthy interruptions of substantive computation will prove to be intolerable for precisely those problems which are now beginning to dominate the interest of the most advanced programming community, hence of the most natural LISP 2 customers. For those problems involve either intimate man–machine interaction (e.g., MATHLAB) or control of elaborate machines (e.g., ROBOT). They are, in other words, problems in real-time computation which may very well founder on interrupts of the indicated magnitude.

A better solution to the garbage-collection problem would have been to break the overall effort of space administration into small quanta and distribute these more or less uniformly over the whole program running time. The main objection to certain existing algorithms of this kind is that they demand explicit, i.e., programmed, erasure of data structures and thus place an unwanted burden on the shoulders of the programmer. But this can be avoided in block-structured systems. In any case, a new way has to be found.

Experience indicates that the solution to the problem of dynamic space administration in a list-processing system so pervades all other systemic issues as to virtually dictate overall system strategy. If this insight has any validity at all, then it would argue that the LISP 2 team should have started with a set of *functional specifications* derived from both the acknowledged power of the LISP concept *and* the environment in which the new system has to operate. Because that environment must of necessity include the facts of real-time application, time-sharing, machines with very large first level memories, and paging, they would have been led naturally to research on distributive compacting garbage collectors. Under such a regimen no implementation would have been started until that major problem was solved. Judging by the superb quality of the individual team members, it is hard to believe they would not have succeeded.

That LISP 2 permits a variety of data types and even the introduction of new data types can no longer serve to distinguish it from competing languages. AED and PL-I, to name only two, have similar facilities. And both permit unlimited recursion and partial word operations. And it has been shown that pattern-driven data-manipulation functions may be introduced into a large class of languages.

LISP's remaining claim to distinction is that "it is unique among programming languages in the ease with which programs can be treated as data." Of course, many languages can "treat programs as data." Even lowly FORTRAN can operate on FORTRAN programs! What is meant here is that a currently *regnant* program can be operated on *and the resulting structure again treated* as a program, i.e., evaluated—all within a single run. In other words, *that data can be treated as program!* (By an unconscionable stretch of the imagination one can conceive of FORTRAN achieving this mode of operation by means of overlays and chaining. That would hardly qualify as effective computing.)

This important feature has been kept in LISP 2, at a price, however, that goes some way toward defeating the "inconvenient language" component of the initial motivating argument. For programs that are to be subjected to such treatment must be in the intermediate, i.e., LISP 1.5-like language. The programmer who wishes to take advantage of this aspect of the power of LISP 2 must, therefore, be conversant in the intermediate language as well as in the ALGOL-like language.

It was said earlier that LISP 1.5 permits constructs of the following type:

$$((fa)b)$$

where f is a function to be applied to the argument a. The results of this application are again a function (as opposed to a function name) which is then applied to the argument b. Thus, for example, if

$$f = (\text{LAMBDA}(G)(\text{LAMBDA}(X)(G(GX))))$$

and

$$a = \text{SQRT}$$

then the result of applying f to a is the function

$$(\text{LAMBDA}(X)(\text{SQRT}(\text{SQRT } X)))$$

which if then applied to $b=16$ will yield the value 2. Had f been applied to the LOG function, it would have yielded the LOG LOG function. This is an enormously powerful feature of LISP 1.5. Unfortunately, the paper under review fails to allude to its presence in LISP 2. Expressions of the kind just illustrated can certainly be written and are undoubtedly legal in the LISP 2 intermediate language. There is a significant question, however, over the possible implementation of such a mechanism within the LISP 2 framework. In LISP 1.5 the interpreter is always kept in core (even when dealing with compiled LISP 1.5 programs) to deal with just such issues. As there is no LISP 2 interpreter, the system must call the compiler whenever such an issue arises during the running of a program. This carries with it the two disadvantages that 1) considerable memory space must be left for the compiler (unless overlay techniques are used!) *and* 2) that non-negligible computing time is once again devoted to purely administrative matters. The overall effect of these penalties is to encroach on the freedom with which the LISP 2 programmer will use this powerful device. And the worst consequence of that is that it increases the burden of awareness which the programmer must carry. It is arguable that, at least for the problem classes relevant to this discussion, the ultimate limit of what can be programmed is determined by the weight of just this burden.

All the above views taken together point to the conclusion that the child has overcompensated for the reputed deficiencies of the father. ALGOL *is* a more convenient language than LISP 1.5 and disparate data types *are* highly desirable. But their inclusion has been achieved at the cost of weakening some of the muscles for which LISP is most deservedly famous.

The very caliber of the LISP 2 architects and builders give the disparity between the hope and the achievement a paradoxical character. How is it to be explained? It was stated earlier that the effort should have been grounded on a set of functional specifications and tuned to the emergent computing environment. What fault there is, is probably attributable to the atmosphere of *development* (even production) in which the task was executed. Under that kind of pressure, the truly elegant foundation of the LISP idea eroded. Then too, it should be appreciated that in comparison with LISP, ALGOL, however practical it may be, is a jungle of ad hoc rules and devices. No fine-spun creation could hope to preserve its charm in such surroundings.

But the wake over lost innocence must not be overly prolonged. The technical achievement of the LISP 2 group deserves the most keen respect and admiration. To say the system rests on nonparsimonious foundations is not to say that it is a patchwork. It is an integrated machine of tremendous versatility. If it lives up to its specifications, it should easily dominate such other eclectic systems as FORMULA ALGOL, FORMAC, and the various SLIP embeddings. Because of the huge advance sale of tickets to the PL-I extravaganza, there can be no contest between it and LISP 2. Whatever LISP 2's performance in popularity polls might eventually be, its ultimate significance must be judged in terms of its influence. This is the measure that established the fame and honor of its direct antecedants. The problems alluded to here wait to be solved. Perhaps the very people who created this system—and who once more demonstrated that a platoon of superior craftsmen is enormously more powerful than a brigade of plodders—may now meet the challenge they have so well illuminated.

JOSEPH WEIZENBAUM
Mass. Inst. Tech.
Cambridge, Mass.

R67-23 A Theory of Computer Instructions—W. D. Maurer (*J. ACM*, vol. 13, pp. 226–235, April 1966).

The research reported on in this paper is, in a sense, an attempt to bring the mathematical formalisms and techniques which have been used for a number of years in research in automata theory to bear on machine models which more closely resemble actual computers. Thus a definition of a particular model of a computer is a 4-tuple of sets $(M, B, \mathcal{S}, \mathcal{I})$. The members m of M represent the memory storage devices in the computer, each of which can assume a value of b, where b is an element of B. An element S of $\mathcal{S}$ represents a possible state of the machine and consists of a mapping of M into B. An element I of $\mathcal{I}$ represents an instruction in the repertoire of the machine and consists of a mapping from $\mathcal{S}$ into $\mathcal{S}$.

The author studies the properties of instructions in terms of their input and output regions. Roughly speaking, the output region of an instruction is that subset of M which can be modified by the instruction, and the input region is that subset of M from which the instruction draws its information. Thus the instruction ADD Y, which adds the contents of memory cell Y to the contents of the accumulator, has cell Y and the accumulator as the input region and the accumulator as the output region (there is an error on line 7 of page 228 of the paper concerning this example). An even more detailed look at the action of an instruction is provided by looking at subsets of these regions. Thus some subset of the input region of an instruction affects some subset of the output region.

Many of the results are concerned with the relationships between the input and output regions of an instruction and their subsets under different assumptions about the computer. The composition and decomposition of instructions as well as product and restructured machines are also considered. Some of these latter results are analogous to results described by Rabin and Scott[1] concerning finite automata.

The paper is a welcome addition to the growing body of research which is attempting to formally model and study computer systems. Other contributions in this area have been made by Elgot and Robinson[2] and Orgass.[3] The presentation is generally clear. The following typographical error was found in the middle of page 231. The description of the mapping performed by an instruction of the product machine should be:

$$I_1 \times I_2 : \mathcal{S}_1 \times \mathcal{S}_2 \rightarrow \mathcal{S}_1 \times \mathcal{S}_2.$$

A. J. BERNSTEIN
General Electric Company
Schenectady, N. Y.

R67-24 Formal Properties of Grammars—Naom Chomsky (*Handbook of Mathematical Psychology*, R. D. Luce, R. R. Bush, and E. Galanter, Eds., vol. 2, New York: Wiley, 1963).

The original development of the theory of formal languages came from the field of linguistics. Since the theory was first developed it has attracted the attention of researchers in programming theory. The reason for this is that some programming languages, for example ALGOL, can conveniently be defined by using a set of rewrite rules or grammar to help specify the form of programs of the language. This has led to the concept of a syntax-directed compiler in which the compilation process is carried out in the following two steps. First, a program is parsed according to the grammar for the language just as one would parse a sentence of a natural language. Then output code is produced using the parse which contains the necessary structural information about the program. Although this method of compiling is generally slower than others, it has the advantage of simplifying the writing of a compiler. Furthermore, once a general parsing algorithm has been written, it can be used for different languages by supplying it with suitable sets of rewrite rules.

In addition to its influence on compiler techniques, the theory of formal languages has aided the development of automata theory. The reason for this is that the classes of languages defined in terms of rewriting rules correspond to the classes of language (sets) accepted by certain types of restricted, infinite state automata. The fact that such a close relationship exists between these two apparently remote areas provides considerable motivation to examine the subject of formal languages in its own right, apart from its original setting in the field of linguistics.

The article reviewed here is a survey of formal languages from a linguistic point of view which was written in 1963. It is actually one chapter from Volume 2 of the *Handbook of Mathematical Psychology*. However, it is sufficiently self-contained to be read as a complete entity although there are references to material in other chapters. The article is now three years old and was not written with computer interests in mind. Nevertheless, it is still probably the best source for those who wish an extensive introduction to the field.

The survey starts with a review of finite automata, pushdown automata, linear-bounded automata and Turing machines. Various rewriting systems are then defined. The class of unrestricted rewriting systems is shown to be equivalent to the class of Turing machines in that the unrestricted rewriting systems generate precisely the class of recursively enumerable sets. The class of context-free languages is shown to be precisely the class of sets accepted by nondeterministic pushdown automata, and the class of languages for which there exists a non-self-embedding context-free grammar is precisely the class of regular sets. Since the book was written it has been shown[1] that the context-sensitive languages are precisely the class of sets accepted by linear-bounded automata.

The survey continues by discussing various properties of the classes of languages. The section on closure properties is concerned with whether or not a given class of languages is closed under operations such as union, intersection, complement, or complex product.

[1] M. O. Rabin and D. Scott, "Finite automata and their decision problems," *IBM J. R&D*, vol. 3, pp. 114–125, 1959.
[2] C. C. Elgot and A. Robinson, "Random-access stored-program machines, an approach to programming languages," *J. ACM*, vol. 11, pp. 365–399, 1964.
[3] R. J. Orgass, "A mathematical theory of computing machine structure and programming," General Electric R&D Center, Schenectady, N. Y., Repts. 66-C-310 and 66-C-311, 1966.

[1] S. Y. Kuroda, "Classes of languages and linear-bounded automata," *Information and Control*, vol. 7, pp. 207–223, June 1964.

The answers are important since they help to characterize the various classes and also to establish other results.

Another section is devoted to transducer properties. A finite transducer is a finite automaton which produces an output (possibly null) for each state transition. Thus a finite transducer maps one set of strings onto another. Given a finite transducer T, we can always find an inverse transducer T^*, such that T^* maps a sentence y to a sentence x just in the case that T maps x to y. The class of regular sets and the class of context-free languages are closed under finite transducer and inverse finite transducer mappings. This is extremely useful in studying the structure of programming languages since one can map a programming language onto a simpler language which contains only the essential structure. By considering only the essential structure one can place a lower bound on the necessary generative power needed to represent the language.

A third section considers decision problems. It turns out that many problems such as determining whether two arbitrary context-free grammars generate the same language or whether an arbitrary context-sensitive grammar generates any sentences at all are recursively unsolvable in the sense that there can be no algorithm that will solve the problem in every case. Most results concerning decision problems about languages that are known today are contained in the survey.

In addition to the above material there is considerable information on such topics as ambiguity in context-free languages, various types of restricted context-free languages such as linear, metalinear and sequential languages, the relation between languages and formal power series, and categorical grammars.

In short, the article contains most of the important results known today and serves as a very useful starting point for anyone interested in formal languages. Those interested in pursuing the subject further should consult the references in the survey, and in Ginsburg.[2]

JOHN E. HOPCROFT
Princeton University
Princeton, N. J.

[2] Seymour Ginsburg, *The Mathematical Theory of Context Free Languages*. New York: McGraw-Hill, 1966.

R67-25 A Syntax-Oriented Translator—Peter Zilahy Ingerman (New York: Academic Press, 1966).

As its title suggests, this book (or more appropriately, monograph) is a detailed description of a specific syntax-oriented translator (basically PSYCO, the Princeton Syntax Compiler[1,2]) rather than an exposition of syntax-related techniques or syntax directed compilers in general. It is, by the author's admission in the preface "vague, obfuscatory, and hubristic." The intended audience is not clear, but it would probably be of interest as supplementary reading for advanced computer science students. Its utility as a text is seriously impaired by lack of exercises, and its utility as a guide for compiler experimentation is impaired by omission of complete translation tables for anything but the conventional assignment statement. Although the author recognized this shortcoming in the footnote on page 21, he failed to produce any real substance in later chapters.

One annoying feature of this book is likely to be the author's invention of new terminology and symbolism in lieu of more accepted terminology and symbolism. Three important instances are: *accretion* for *string*, *object* for *character*, and *unparsing* for *generation*. In addition, the prefix *meta* is used excessively, even when no confusion from lack of formal stratification would result.

[1] E. T. Irons, "Maintenance manual for PSYCO—part one," Institute for Defense Analyses, Princeton, New Jersey.
[2] E. T. Irons, "A syntax directed compiler for ALGOL 60," *Commun. ACM*, vol. 4, pp. 51–55, January 1961.

Chapter 1 reads like an advertisement for syntax-oriented translators, and indulges in exaggeration, misleading inference, and selective omission. For example, the comparison beginning on page 17 suggests that it is a simple matter to produce syntax tables which are both error-free and "achieve any desired balance between translation time and running time." In fact, producing and debugging syntax tables for a nontrivial language is definitely nontrivial. Furthermore, the type of code improvement possible in syntax-oriented translators of the type described in this book is limited to improvement in local expression evaluation. In truth, the principal value of syntax-oriented translators is in their use as research vehicles as stated on page 19. The book, however, would have been improved considerably by some complete examples of nontrivial research use.

In Chapter 5, the author observes that real languages have pathological features which must be handled by ad hoc extensions to a purely syntax-oriented translator. He then offers a number of such ad hoc extensions to solve problems that have occurred to him. Not included are, for instance, a facility for handling attributes of symbols, efficient subscripting, end-of-line and continuation conventions (mundane but important), FORTRAN equivalence processing, PL/I and COBOL data structures, PL/I and ALGOL block structure, fixed format input, mapping of long identifiers onto assembler acceptable identifiers, diagnostics and error recovery procedures, storage allocation, and constant conversion.

One major problem with the technique presented is that the entire source text must be parsed before any generation can begin. This severely limits the size of programs which can be translated, even on a machine with large memory. The technique presented also pays a penalty in efficiency by building trees depicting the inner structure of identifiers and constants. In the principal example of the book (the parse of "$A=B/(C+D)$"), one-fourth of the nodes in the parse tree are required merely to turn the letters A, B, C and D into identifiers. If the identifiers had been three letters long, over half the nodes in the tree would have been so used. This is clearly the time for another ad hoc extension.

Failure to discuss ways of handling illegal input strings (or accretions) is more insidious. Almost all programs, in fact, are either syntactically or semantically incorrect when first compiled. Any compiler, even a syntax-oriented one, should do better than print, "INPUT ACCRETION UNGRAMMATICAL."

This paragraph is a collection of miscellaneous thoughts while reading the book. The section on canonical ordering seems ill motivated. There is much ado about a linear ordering of a grammar for parsing, when, in fact, it can easily be built as a linked list. The book condemns the "left recursion" problem of "top down" analyzers, when left recursion is as simple to detect and remove as "cyclic nullity" in "bottom-up" recognizers. Do loops can be handled much more simply than the technique in the book by the addition of another push down list. In all of the grammar tables, space could be saved (and clarity improved) by assuming that the reader knows the letters and digits. Finally, although usage of abbreviations in grammar tables is reasonable, reduction of names to two letters makes them unnecessarily cryptic.

Without detail auditing of flowcharts, they do appear adequate to allow a determined and persevering reader to actually construct a processor. Although it is doubtful that many readers will undertake to write one, it is to Ingerman's credit that he put this information in a permanent and accessible place. Although this book has shortcomings, at least it was written, and it does attempt to deal with a difficult subject plagued by a paucity of publication.

Finally, for those who require a review to contain at least some pedantic "nit-picking," we observe that the last paragraph refers to a capital letter as a majuscule and a lower-case letter as a minuscule, and in the process misspells both words.

ROBERT M. MCCLURE
Texas Instruments
Dallas, Tex.

E. Graphical Display Devices

R67-26 The Lincoln Wand—L. G. Roberts (*1966 Fall Joint Computer Conf., AFIPS Proc.*, vol. 29, pp. 223–227).

Most computer-driven graphical display devices are capable of displaying in only two dimensions. However, many systems have been written which can display three-dimensional objects in two-dimensional projection. By allowing the observer to manipulate (especially rotate) the object being displayed, these systems allow the observer to obtain a clear mental image of the three-dimensional object being displayed. Thus problems involving the design or visualization of three-dimensional objects often can be solved with the help of an interactive computer-driven display.

One of the most difficult problems in three-dimensional interaction is to provide a natural way for the man to "talk" to the machine about the object. Usually a position- or point-sensing device (e.g., RAND Tablet or light pen) is used, sometimes with the addition of coordinate-controlling knobs, a "joy stick," or a rotatable sphere to control rotation of the object being displayed. None of these methods is really satisfactory to the man observing or controlling the display.

The Lincoln Wand, described in the article being reviewed here, is the first device which provides direct three-dimensional input from man to computer. It is intended to be used in conjunction with a display console, much as a light pen is used. It differs from a light pen primarily in that the computer can sense the position of the Lincoln Wand within a volume, rather than just on a surface. A model of the Lincoln Wand has been built and is operating at the Lincoln Laboratories. It appears to be accurate, fairly simple, very flexible, and not too expensive.

The Lincoln Wand consists of four pulsed ultrasonic transmitters, located at the corners of a rectangle which can be flush with the front panel of a display scope, and an ultrasonic receiver in the end of a pen-shaped hand-held object (the "wand"). It operates by measuring the transmission times of sound pulses from each transmitter to the wand. The installation at the Lincoln Laboratories has an absolute positional error of about 0.2 inch and a positional stability (short-term accuracy) of 0.02 inch. The latter is the more important in man-machine communications. The working area of the wand is incredibly large: 4 by 4 by 6 feet.

One problem with the Lincoln Wand is that a direct sound path must exist from each of the four transmitters to the wand. If it does not, the wand may detect an echo pulse rather than the direct pulse. (The use of four transmitters rather than three allows this error to be detected, and the resulting position point to be ignored.) Thus the "free space" in which the wand operates must be free of obstructions. This would seem to make it difficult to use the wand to point to physical objects or to trace surfaces.

D. R. Haring of M.I.T. has suggested[1] that a "pen" be built using accelerometers to provide inertial sensing of pen position. Although in his memorandum he proposes only two-dimensional sensing, he sees no reason why the method cannot be extended to three-dimensional sensing if the problem of correcting for gravity is not too severe.[2] Such a device would be free of the "free space" restriction of the Lincoln Wand.

The existence of a tool such as the Lincoln Wand should result in new applications for computer-driven displays. In addition to its obvious value as a hand-held drawing and pointing instrument, it should be a useful computer input device when attached to any object the computer must track or control.

Now that a solution exists to the problem of three-dimensional input, it seems appropriate to consider three-dimensional displays. It would be fairly easy to generate a stereo pair of images of an object, if there were a convenient way of coupling the display to the observer. Alternately, it might be possible to devise a holographic display device. The Lincoln Wand, together with an effective three-dimensional display, would provide a degree of man-machine coupling far exceeding anything that has been achieved to date.

Rudd H. Canaday
Bell Telephone Labs. Inc.
Murray Hill, N. J.

[1] D. R. Haring, "Proposal for an inertial pen as a computer input device," M.I.T., Cambridge, Mass., Memorandum MAC-M-322, August 16, 1966.
[2] ——, private communication.

F. Analog and Hybrid Computers

R67-27 An Automatic Acoustic Ray Tracing Computer—L. Light, J. Badger, and D. Barnes (*IEEE Trans. on Electronic Computers*, vol. EC-15, pp. 719–725, October 1966).

The "computer" is a small, desk-top, 10-volt, solid-state, transistorized analog computer, to which has been added two banks of stepping switches. The latter serve to control four banks of pots, which are the equivalent of four arbitrary function generators. This computer is used to plot traces for shallow underwater sound rays. Reflections from the water surface and an arbitrary ocean bottom are allowed. The sound-velocity gradient is taken as a constant in each segment of the profile. (It was not clear from the paper whether the sound-velocity profile was also taken to be a constant for each segment.)

The ray computation takes three minutes, which the authors consider to be "rapid operation."

Within the framework outlined above, the authors have succeeded in producing an inexpensive analog ray tracing computer, which they say is also useful.

By confining their attention to shallow rays only, not providing for variation of the sound velocity profile with range, and not constraining the ray to satisfy Snell's law, they have produced a computer program which can be set into a small analog computer. This program is a simplification of a more general program reported by Rubin and Graber.[1]

The authors have made three approximations to the exact differential equation, two of which alone would make the acceleration term too small, while the third approximation alone would make it too big. All three taken together, however, produce a final equation which is surprisingly accurate. The authors did not say how accurate, but I have calculated that their final equation is probably accurate to within 0.02 percent for shallow rays whose path angle does not exceed $\pm 15°$.

As the authors correctly state, one of the principal factors limiting the accuracy of the solution is the precision of insertion of the "bottom" profile. Many users of computers in other fields overlook this common limitation (the precision of the knowledge of some crucial input function, or constant, to the significant accuracy of the computer solution).

The authors might find it interesting to attempt to trace a channel ray, assuming an infinite bottom, for an initial state which will cause no surface reflection, as in Figs. 11 and 12 of Rubin and Graber. The individual peaks and valleys should occur at the same values of depth from cycle to cycle, as in Fig. 12. This is the ultimate test of the accuracy of the solution. However, it is this reviewer's belief that the solution will either diverge, as in Fig. 11, or converge, depending upon the characteristics of the computer used, because the ray is unconstrained.

The authors could make a real contribution to the ray-tracing art by substituting high-speed electronic gates for all mode and other relays, so that they could then produce a ray in about 100 ms. In this case, the computer could be run in high-speed rep-op, allowing the ray to be displayed on an oscilloscope. The analyst could then *instantly* perceive the effect of small changes in the parameters of interest. In this manner, the analyst could obtain ensembles of solu-

[1] A. I. Rubin and G. F. Graber, "Acoustic ray tracing on the general purpose electronic analog computer," *IEEE Trans. on Electronic Computers*, vol. EC-14, pp. 443–455, June 1965.

tions, where "footprint" curves,[2] showing, for example, the variation in final travel time differences as a function of initial ray angle, initial depth, or any other parameter, are plotted, rather than individual trajectories.

Finally, this application shows that the analog manufacturers have continued to neglect the problem of quick, easy, reliable changing and inserting of arbitrary functions for the low-cost computer. They have solved this for the high-cost computer with elaborate card set diode function generators. Unfortunately, one of these devices may exceed the entire cost of an "inexpensive" analog computer.

ARTHUR I. RUBIN
Analog/Hybrid Computations
Martin Marietta Corp.
Baltimore, Md.

[2] A. I. Rubin and L. Shepps, "A general purpose analog translational trajectory program for orbiting and reentry vehicles," *1966 Fall Joint Computer Conf., AFIPS Proc.*, vol. 29, pp. 771–788.

R67-28 A General-Purpose Analog Translational Trajectory Program for Orbiting and Reentry Vehicles—A. I. Rubin and Lloyd Shepps (*1966 Fall Joint Computer Conf., AFIPS Proc.*, vol. 29, pp. 771–788).

This paper is a description of the authors' application of high-speed computation methods to the analysis of several aerospace problems. The examples shown include several parametric studies of reentry from orbit, including plots of entry angle against thrusting angle with incremental deorbit velocity change as a parameter and "foot print" plots with lift-to-drag ratio as a parameter. The analysis method features high-speed analog computer solution of the translational differential equations of motion of aerospace vehicles. Suitable equations and flow diagrams for the trajectory computation are presented. The high-speed trajectory program, coupled with automatic parameter variations and a final-value plotting routine, permits rapid economical parametric studies for engineering analysis purposes.

The paper is well written, the figures are adequate and no factual errors were found.

The title and text of the paper refer only to analog computation but this reviewer would prefer to call the methods used here "hybrid"—a term not yet well defined when applied to machine computation. This point will be discussed later in the review.

The analog—or hybrid—methods used generally are not new and the problems treated have been solved by other methods, so that one might be tempted to label this paper a routine application report. This is not true, but unfortunately the reasons why it is not true are concealed in the cost and solution-time figures furnished by the authors. It is difficult to realize fully, until one actually witnesses the computer in operation, that accurate high-speed analog computation automatically controlled by the results of high-speed digital logic operations indeed is a new field of computation.

The applications discussed in this paper lie within that field of problems which are discouraging, to say the least, due to the sheer bulk of calculations needed and the consequent long computation time when one uses slower methods. By utilizing the extremely high information-processing rate of the parallel analog computer to handle most of the calculations, many problems which in the past have been considered impractically long, can be analyzed conveniently and economically. Since this fact has not yet been widely exploited—or even realized by many engineering analysts—the paper is valuable as a practical example of what can be accomplished by straightforward hybrid computation methods.

As mentioned earlier the reviewer prefers to call the methods used in this paper "hybrid" rather than analog. The term analog is still too closely associated with well-remembered difficulties with servo-multipliers and poorly-stabilized operational amplifiers, for one thing. More importantly, the fairly recent commercial availability of accurate, fast electronic analog computing elements, high-speed individual integrator mode switching and an associated complement of digital logic elements makes the modern "analog" computer a much more accurate, reliable, and, above all, versatile machine than many analysts realize.

The definition of a hybrid computer which includes any analog with an associated digital operation—a switch—seems much too broad. The definition which includes only general-purpose analog machines tied together with general-purpose digital machines through A–D and D–A converters seems too narrow. A not unreasonable definition would require at least individual integrator mode control and some modest complement of digital logic elements.

There seem to be two distinct types of hybrid computation emerging: one might reasonably be called parallel, the other serial or sequential.

Parallel hybrid computation features an analog computer and a digital computer, usually linked with conversion equipment, performing similar tasks in parallel. The analog performs high-speed limited-accuracy operations and the digital simultaneously performs lower-speed higher-accuracy operations, the two computers cooperating to produce a faster, more accurate solution than either could alone.

Serial, or sequential hybrid computation features alternate, or sequential operation of the two computers. It is characterized by high-speed analog computation (mainly integration) of a result, which is transferred to a digital computer or set of digital elements. The digital computer performs mostly algebraic and logical operations on the analog result and transfers the logic results to the analog in the form of mode commands, parameter variations, etc.

The sequential hybrid category has enormous potential as an engineering and scientific analysis tool.

The paper being reviewed is a good example of the application of elementary sequential hybrid computation methods. In this case, a fairly elaborate analog computer was used, with only a few digital logic elements performing the required decision operations.

L. E. FOGARTY
University of Michigan
Ann Arbor, Mich.

R67-29 Hybrid Computer Solutions of Partial-Differential Equations by Monte Carlo Methods—Warren D. Little (*1966 Fall Joint Computer Conf., AFIPS Proc.*, vol. 29, pp. 181–190).

Monte Carlo solution of elliptic and parabolic linear partial differential equations was first suggested by Von Neumann and Ulam in the early 1940's. As a simple example, one can solve Laplace's differential equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{1}$$

on a plane region D bounded by a suitable curve C with given boundary conditions

$$u(x, y) = u_c(x, y) \quad \text{on } C \tag{2}$$

by generating a random walk with the solutions $x(t)$, $y(t)$ of the stochastic equations of motion

$$\frac{dx}{dt} = X(t) \qquad \frac{dy}{dt} = Y(t) \tag{3}$$

with starting values $x(0) = x_0$, $y(0) = y_0$, and independent white-Gaussian-noise forcing functions $X(t)$, $Y(t)$ with zero mean and unit power spectral density. These random walks will cross the boundary C at random points x_c, y_c such that

$$E\{u_c(x_c, y_c)\} = u(x_0, y_0).$$

Hence, the sample average $u_c(x_c, y_c)$ over a sufficiently large number n of random walks is an unbiased estimate for the solution $u(x_0, y_0)$ at x_0, y_0. The method is particularly intriguing because the procedure is not significantly more complicated in the three-dimensional case.

Because of the relatively large number of random walks required (on the order of a thousand), digital-computer Monte Carlo computation has not been competitive with finite-difference methods. Chuang, Kazda, and Windeknecht, at the University of Michigan, showed the feasibility of implementing the random walks on an analog computer solving the stochastic equations of motion (3) for some simple boundary problems, but it remained for Dr. Little's thesis to extend the analog/hybrid method to a wider class of parabolic as well as elliptic differential equations; the random walks were implemented on a conventional "slow" analog computer requiring about five minutes for a thousand random walks. A digital computer also controls analog computer operation.

Dr. Little's outstanding paper (it was awarded the $500 AFIPS prize for the best 1966 FJCC paper, and an earlier paper based on Dr. Little's thesis won a Simulation Council student-paper award) contains significant contributions to the practical hybrid-computer solution of parabolic differential equations. This work has already formed a useful basis for a follow-on project employing a faster analog computer (a thousand random walks per second) and analog averaging in the reviewer's laboratory. Dr. Little's complete thesis, completed in 1965 at the University of British Columbia under the supervision of Dr. A. Soudack, is recommended reading for workers in this field, since it contains much practical and useful information on noise generation, calibration of computer setups, suggestions for future work, etc. Last but not least, Dr. Little and his co-workers are to be congratulated for the actual physical implementation of this project with a 231-R analog computer, homemade linkage, and a vacuum-tube-type digital computer.

GRANINO A. KORN
University of Arizona
Tucson, Ariz.

R67-30 Analog Computer Methods for Parameter Optimization—R. A. Harvey, G. R. Taylor, and R. D. Benham (*Simulation*, vol. 6, pp. 181–191, March 1966).

This paper is concerned with the use of analog computers for optimizing parameter values in mathematical models of physical systems. Unfortunately, the authors apparently are not familiar with gradient methods and their mathematical properties. Consequently most of this paper is devoted to special techniques used with specific applications and with little generality.

Throughout the paper there is a confusion between minimization of functions in a mathematical sense (which requires the location of points at which derivatives of appropriate distance functions become zero) and feedback control system behavior. For example, the first portion of the paper is concerned with fitting the solution $y_m(t)$ of a model equation with the solution of a system equation $y(t; p_1, p_2)$ where p_1 and p_2 are the two parameters to be selected in an optimum manner. The authors claim that either the error $(e=y-y_m)$ or its integral $(f=\int_0^T e dt)$ can be used as a "performance index." This is simply not true in general, since neither e nor f is a distance function. In general, neither e nor f has finite minima. The authors' computer circuit works because: a) they have selected to study only the step response of a first-order linear system, which is strictly monotonic, and b) they have complete a priori knowledge of both model and behavior, including the parameter influence coefficients. (For this particular system, it can be shown that the procedure in the paper is equivalent to an approximate gradient procedure based on the criterion function $I=\int_0^T |e| dt$, which *does* possess the properties of a distance function.) Evidently, since complete knowledge of the answer was required beforehand, the technique, as explained in the paper, has no practical usefulness whatsoever in system identification.

The second portion of the paper describes a model for a uranyl nitrate extraction column. Fortunately, an integral squared error is used as the performance index in this case. The optimization consisted of automatic plotting of criterion surface contours and selection of the best value by inspection. This technique is not practical with more than two parameters.

An example of automatic optimization by sequential random perturbation is then presented. Since the equation being studied is algebraic, such optimizations are better done on digital computers. No reference is given to the published material on the convergence of random search optimization techniques and their modifications (such as reported by Korn and others).

The final example in the paper is a rather interesting nonlinear model of river flow following upstream changes in flow rate. The technique presented is basically a discrete gradient method using an integral squared-error performance index. The authors indicate parameter cross-coupling as a problem, but again offer no indication of convergence problems, the choice of optimum step size (determined by adjustment gain in their circuit) and other problems treated by other authors.

To this reviewer, this paper is so shallow in its treatment and so specific in its discussion of examples that its usefulness to other computer users is severely limited.

GEORGE A. BEKEY
Dept. Elec. Engrg.
University of Southern California
Los Angeles, Calif.

R67-31 Transfer Functions and the Matric Computor—Pierre M. Honnell and R. J. Mulholland (*Proc. Internat'l Assoc. for Analog Computation*, vol. 8, April 1966).

The article presents the "Matric Computor" as " a revolutionary new analytic computing device" and its "application to . . . problems described by transfer functions." It is further said, in the conclusion, that "Inevitably, the Matric Computor will completely supplant analog computors by whatever name, and, in many cases . . . displace arithmetic machines."

The unaware reader is by all means intrigued by such extreme statements, and proceeds to reread the article a second time to find out what is so outstanding in this new computer:

The Matric Computor, as described in this article, is an analog machine, organized systematically in a matrix structure, in such a fashion that linear algebraic and differential problems can be programmed with great ease.

This being the limit of capabilities of the machine described in this article, it immediately appears as a special purpose linear analog computer.

Special-purpose computers usually have a superiority in the solution of problems within their limitations. However, techniques for solving linear differential problems using general-purpose analog computers are well known, and it was not apparent to the reviewer that the Matric Computor did present any drastic improvement in programming over general-purpose machines.

In brief, the article fails to convince that the Matric Computor is as outstanding as its authors claim it is, and leaves the reader with more than serious doubts in mind.

Could the authors of a paper published in 1966 ignore that most problems attacked by analog computer these days are nonlinear, therefore outside of the reach of the Matric Computor, and really believe that "analog computers by whatever name" will be inevitably completely supplanted by the Matric Computor?

That the Matric Computor is an easy-to-use, rather specialized analog computing device is certainly the case; but, by claiming universal and absolute superiority, the authors certainly present a picture which is difficult for the professionals of analog computation to accept.

ROBERT VICHNEVETSKY
Electronic Associates, Inc.
Princeton, N. J.