Computer Science
Research Review

1966

Computer Science Research Review

1966

An Annual Report
published by
Carnegie Institute of Technology
Computation Center and
Department of Computer Science

Edited by Joyce Nissenson

The work reported here was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense: Contract SD-146, also supported in part by a grant from the Mellon family and the Richard King Mellon Foundation.

Pittsburgh, Pennsylvania
1966

Computer Science
Research Review

Table of Contents

It was the best of times, it was the worst of times,
it was the age of wisdom, it was the age of foolishness,
it was the epoch of incredulity, it was the season of
of Light, it was the season of Darkness, it was
the spring of hope, it was the winter of despair, we had
everything before us, we had nothing before us,
we were all going direct to Heaven, we were all going
direct the other way—in short, the period was so far
like the present period, that some of its noisiest
authorities insisted on its being received, for good
or for evil, in the superlative degree of comparison only.

Charles Dickens
A Tale of Two Cities

Introduction

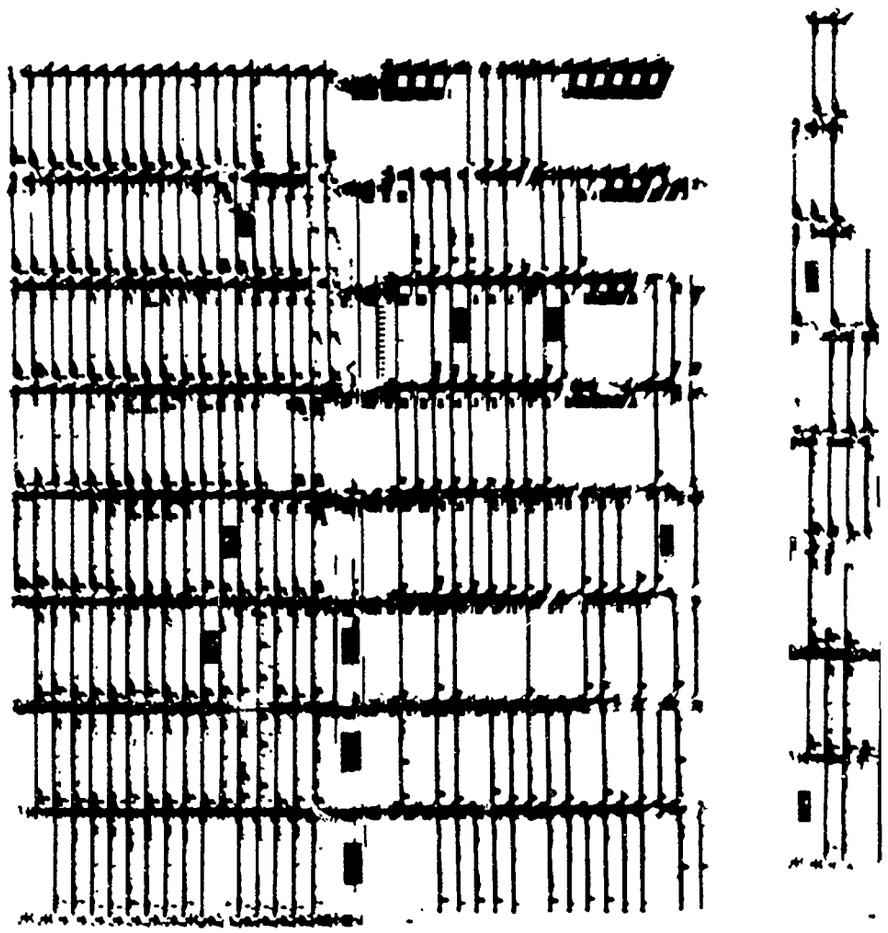
This document tells something about research in information processing at Carnegie Institute of Technology in 1966. It tries to say it mainly by a series of essays, written by some of us in the environment, that reveal aspects of computer science of concern here. Although we have included a certain amount of descriptive material—listings of people, reports, and so on, we have avoided the long compilation of small paragraphs of progress, common to most progress reports. Such compilations have their uses, but mostly they just create a fiction. They present the appearance of a neat organization or research—here is what is going on at Carnegie (or wherever). But research cannot be so packaged, and the picture of a social process under control is largely spurious. In the main this is because research efforts are related to their ultimate goals by a strong bond of hope, as well as by a weak chain of rationality. In the idiom of problem solving programs, one has at best tests to avoid foolishness. No reliable differences can be had between a current state of

knowledge and a desired one. To be sure, one must move forward and explore. So one picks a goal—one decides to build a new programming language, or to prove that a program does what it claims. But the goal itself is only a surrogate, only a means to an end. There will be no difficulty recognizing the end: the new technique; the new insight into the nature of information processing; the new whatever; each will be clear enough when it occurs (at least to a small subset of the field). But these final results often bear only a tangential relation to the initial surrogate goals.

What, then, can be said about progress? Certainly the scientific papers that have been produced should be put forward. They represent science in units that seem appropriate to the scientists. The public and social character of science says that each piece of work shall be communicated to the field in a packaging of the scientist's own choosing. But beyond that, perhaps a communication whose degree of precision matches the reality is most appropriate. That is what this report attempts to be.

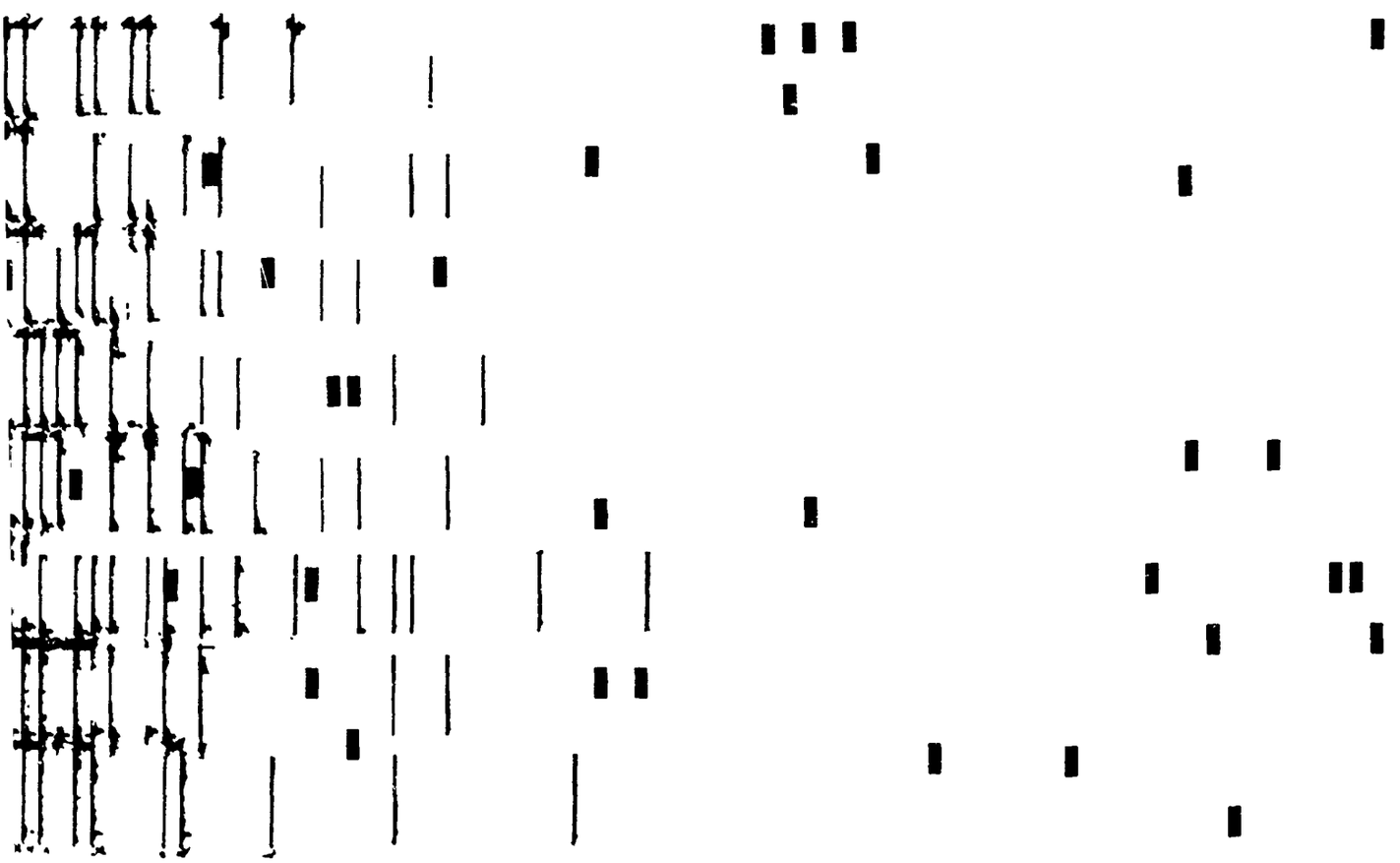
Is it appropriate to infer from these essays the future directions that computer science will take at Carnegie? Yes, as long as one takes into account the selection from the total research activity that has occurred here. Within the Department of Computer Science and the Computation Center there is work on graphics, on proving that programs do what they purport to do, and on systems programming. Moving outward, information processing models of human behavior are a major concern of the psychology department; management information systems is an area of active research in the Graduate School of Industrial Administration; and the development of problem oriented programming languages is going on in several engineering departments. None of these efforts are shown here, except most obliquely.

But selection, stringent as it is, seems to us appropriate, since it permits a glimpse, not only at the content of current research directions, but at the intellectual climate in which all the research at Carnegie is being done.



The problem is not how to produce great men, but how to produce great societies. The great society will put up men for the occasions.

A. N. Whitehead
Science and the Modern World



David H. Nickerson, Director

Managing a Computation Center

The Computation Center at Carnegie Institute of Technology is a formal structure designed to ensure three principal interactions

> To make the work of one information scientist available as a tool for other information scientists.
>
> To provide a facility for educating graduate and undergraduate students in the Information Sciences.
>
> To provide a flexible research tool to other researchers on the Carnegie campus to enable them to exploit the advances in computing achieved here or elsewhere.

The central management problem is to achieve the three principal objectives in an optimum sort of way.

The organizational philosophy proposed borrows heavily from contemporary ideas on business organization. Most persons would characterize our solution as 'business-like'. Perhaps it will be useful to consider a brief description of Carnegie's Computation Center and what we mean by 'business-like' before discussing our organizational philosophy.

Carnegie's Computation Center is intended to operate as a public utility. We maintain the Center for the benefit of a large class of users. In the last year, over 1500 users registered to use the Center. Of these, approximately 500 were pursuing research projects which required the use of a computer for successful completion; 500 were using the computer for course work; and the remaining 500 were a blend of computer scientists (about 130), and users with specific computational needs of an industrial (about 50), academic (about 250). or computing support (about 70) nature. Most of our usage is for compilation and testing of individual programs. Most such programs are written for the purpose of solving a problem or demonstrating a concept. Very few user programs are executed repeatedly over a long period of time; thus, we have few so-called "production runs".

The presence of a large computer science effort on campus, and the commitment to make the results of its research available to the community, leads to a unique set of responsibilities. This commitment leads us to constantly introduce novel features. At the same time, we are to keep the environment relatively stable over the duration of a typical large programming development project (2-4 years). This stability is necessary to avoid reprogramming of large research programs.

Interaction with other scientific communities is a two-way process. Many times other communities will develop tools of interest to our own community. For example, at MIT a language may be designed for talking about a certain class of string manipulation operations. Our community wants to use this kind of language and to teach courses in it, yet it is a non-trivial problem to make available a local version of the language. It is the responsibility of the Computation Center to invest the effort necessary to make a local version available.

To give some appreciation of the type and volume of work done at the Center, two charts reflecting operations are included. Chart A shows the principal languages in use with the percentage of time used by these languages. Chart B shows the various departments which use the Center and their percentage of total usage. An appreciation of the computing scientist's

viewpoint of the Center's responsibilities may be gathered by reading Reference 1, and the objectives of the rest of the community are discussed in Reference 2. When we speak of a computation center, we mean one with similarities in form, size, or function to Carnegie's Computation Center.

We now turn to the more complex issue of what we mean by 'business-like'. We will immediately engender controversy by stating that the concept of business management to be employed is the modern management concept. One of the clearest statements of the managerial dilemma, which leads to the development of control systems for modern business, is given in Alfred P. Sloan's book. He is discussing his analysis of General Motors' failure to see a downturn in demand in the 1924 model year.[3]

> Now what did this tale of internal conflict over statistics come down to? Essentially, it was a matter of statistical controls versus salesmanship, which was brought to a head in 1924 by a recession in the general economy following directly upon the boom year of 1923. At that time, the salesmen and the general managers were caught in the illusion of riding the wave. In our then excessively decentralized scheme, I let them ride. Actually, however, this was not a mere bias in favor of the salesmen, for I had no convincing information with which to counter their intuitions. The information was weak because it was neither accurate nor comprehensive enough. It was arrived at by inference from dealer stocks and unfilled orders. This was good enough over a period of time, but the critical trouble was precisely the length of the period. We knew nothing about the most recent five or six weeks of our car sales, and this gap, therefore, was filled with the speculations of the protagonists —the statisticians with their trend lines on the one hand, and the salesmen with their optimistic intuitions on the other. I was, as I have said, in the middle without any means to judge the contending claims—not a comfortable position for a chief executive officer.

A quarter of a century earlier, Thorstein Veblen had described the emergence of the managerial technology needed to provide the tools Sloan was to implement. After first observing that the classical concept of cause and effect was being replaced by a concept of the interaction of process to produce a result, he analyzed

Chart A

Program Summary for May 1 to May 31, 1966

System Name	Number of Programs	Percent Total Number
ALGOL	9360	46.02
THAT	35	0.17
LIST	33	0.16
GATE	569	2.80
AND	2944	14.47
TIPL	1	0.00
IPLV	697	3.43
JOBCD	33	0.16
TAC	383	1.88
MTHAT	38	0.19
WHAT	1	0.00
SCADS	1383	6.80
MAGIC	5	0.02
XANDU	1	0.00
MONTH	1	0.00
ALIBN	40	0.20
COMET	2	0.01
BOOLE	667	3.28
HANDY	52	0.26
ANDRE	88	0.43
COMIT	6	0.03
UPDAT	232	1.14
TIPLV	422	2.07
FORTN	171	0.84
XTHAT	1	0.00
MNTH	47	0.23
MARK	2523	12.40
NOTE	64	0.31
GAME	3	0.01
FORML	293	1.44
TIME	73	0.36
BILLI	6	0.03
TREWQ	4	0.02
GOGO	45	0.22
SPITE	83	0.41
FAUST	2	0.01
HYDRO	21	0.10
MNTHM	2	0.01
MNTHU	8	0.04
CALL	11316	
TOTAL	20339	

the technician's metaphysics.[4]

> To the technologist, the process comes necessarily to count, not simply as the interval of functioning of an initial efficient cause, but as the substantial fact that engages his attention. He learns to think in terms of the process, rather than in terms of a productive cause and a product between which the process intervenes in such a manner as to afford a transition from one to the other. The process is always complex; always a delicately balanced interplay of forces that work blindly, insensibly, heedlessly; in which any appreciable deviation may forthwith count in a cumulative manner, the further consequences of which stand in no organic relation to the purpose for which the process had been set going.

By 'business-like' we mean the elaboration of the processes required to achieve the objectives of the computation center, the systematic reduction of the interaction of these processes to measurable phenomena, and the controlling of the processes to ensure sensitivity to the original objectives. We also mean the simultaneous elaboration and refinement of the objectives themselves and their interpretation in a changing context.

Our organization at Carnegie is designed to be consistent with the processes which we have designed to achieve our objective. Because we feel that the underlying processes are similar to those underlying large segments of modern industry, our organization parallels industrial organization. There is a production department (Operations), a marketing department (User Relations and Planning), a design department (Programming and Engineering) and a research group (Computer Science Department).

By organizing in such a manner, we are committing ourselves to look at the world in a certain sort of way. The use of words like "control", "process", and "measure" imply that we have certain preconceived notions as to what should be going on. It commits us to an examination of the environment, and the structuring of the environment to facilitate the development of that which we think belongs there. Such a viewpoint creates what one could call a "structured" environment. There are several centers which make a virtue of the lack of a "structured" environment.

To understand the case for the unstructured environment, one needs to be reminded that any time an organization comes into being, there sprouts within it a self-justifying concept, or world view, which strongly reinforces views which are compatible with its continued existence in its present form; and which discourages developments which imply change, however unrelated

Chart B

Typical Monthly Usage for Computation Center

Department	Total No. Programs	Actual Time Rati
Administration	0	0.000
Biological Sciences	17	0.000
Business + Social St	0	0.000
Chemical Engineering	282	0.025
Chemistry	907	0.074
Civil Engineering	1637	0.076
Computer Science	3213	0.077
Electrical Engineering	2646	0.174
English	0	0.000
Fine Arts	41	0.007
Graphic Arts	8	0.000
Home Economics	0	0.000
Industrial Administration	1449	0.107
Mathematics	2055	0.034
Mechanical Engineering	2695	0.173
Margaret Morrison	0	0.000
Metallurgical Engineering	79	0.002
Music	52	0.007
Physics	739	0.047
President's Office	0	0.000
Psychology	247	0.016
Student Affairs	8	0.000
Systems Communications	765	0.040
Computation Center	1386	0.045
Comp. Center Research	1042	0.045
Development Priority	11	0.001
System Maintenance	711	0.043
External	263	0.006
Grand Total	20253	

they are to the stated objectives of the organization. The most positive argument then against the selection and structuring of an environment for research is that the environment in some sense dictates the research undertaken. Thus, in a structured center, some work might not be initiated and other work will proceed along previously trodden paths. Most work at Carnegie, for example, takes place in ALGOL because ALGOL is the best maintained, most flexible language available. Several local programming systems are implemented using ALGOL as a tool, and are limited in some sense by ALGOL. In a completely unstructured environment, these languages might have been implemented some other way, and thus had more flexibility in input/output, more macro capability, more program segmentation capability or, to be general, more potential utility. The point is that they may have been limited by the tools (ALGOL) in the environment.

The most complete answer to such a statement first clarifies that, in any event, the user will use a set of tools to perform his research, and he will seek to use the best available. The presence of a large well-orderec set of tools would not in itself preclude the use of other tools if the researcher would have, under other circumstances, been able to find them. In fact, it is fundamental to our approach in graduate education that the presence of a cohesive set of arguments for a given position is an asset and not a hindrance in generating new research.

This point of view is also examined by Thomas S. Kuhn in his essay, "The Structure of Scientific Revolutions" He first develops the notion of a scientific paradigm. This is a much broader view than our concept of a structured environment, i.e., one in which methods, tools, and the problems to be solved are shared by a community. However, he comments specifically on standard tools and procedures. He first explores the inadequacies which a set of shared tools can lead to and decides that the advantages outweigh the disadvantages. He concludes—

> Ought we conclude from the frequency with which such instrumental commitments prove misleading that science should abandon standard tests and standard instruments? That would result in an inconceivable method of research. Paradigm procedures and applications are as necessary to science as paradigm laws and theories, and they have the same effects.

There are positive arguments for furnishing the research tools. Our own observation is that in an unstructured environment, he will first spend considerable time selecting his tools before performing his research. This approach requires that the researcher be abreast of the programming technology in the areas in which he will need tools, or that he be content to use methods which could be inappropriate, inefficient, or obsolete. He also commits himself to a preliminary testing and refining of tools finally selected. This would be satisfactory in an environment where tools work under a wide variety of environmental conditions. This is not the case in the Information Sciences. The tool exists for some time before it is published. When it is published, it is generally not completely described and, all too frequently, it will run on only one particular machine or type of machine, and requires a significant amount of work to become useful on another machine. It seems as likely that the user will bog down due to the lack of a particular tool, or use an inadequate tool, in an unstructured as in a structured environment. This is particularly true if the structured environment recognizes that one of its responsibilities is to maintain adequate facilities for the "maverick" user who does not feel that the officially maintained and developed tools are adequate for his use. Users must be allowed the freedom to "suffer" outside the maintained environment as long as they are aware of the price and are willing to go it alone.

A second major argument, which we have advanced previously[2], concerns the stability of the unstructured environment. It has been our observation that the principal defense against the inadequacies detailed in the preceding paragraph has been the use of the researchers themselves to do systems work, thus creating a defacto structured environment. The difficulty with this approach is first, that the researchers tend to become service agents instead of researchers and, second, that many times systems become unserviceable for long periods of time due to the absence of their sponsors. Also, we have noticed a tendency in the

researcher to become "committed" to his approaches because of the tremendous amount of effort required to generate a new approach. Finally, inevitably the researcher "freezes" his tool development, and thus ends precisely where he did not want to be in the first place.

The above arguments have concerned themselves primarily with fundamental intellectual arguments against structured research environments. There is also a variety of organization which is rigidly structured, but which does not overtly concern itself with the wider issues of its overall objectives. Generally, such a center becomes rigidly fixed on the achievement of one or two relatively minor goals and, as a result, fails to contribute constructively to its environment. One center might, for example, emphasize capacity, another become the promulgator of a particular language or technique at the expense of its mavericks, yet another center will attempt to eliminate all overhead expense. Each of these variants of the structured approach reflect management shortcomings which are well understood by the more successful modern day businessman. The principal weaknesses in all such endeavors is the way chosen to organize them.

It will be useful in drawing an analog to the industrial world to find a close parallel between our activities and those of industrial organizations. Several comparisons suggest themselves. Many see the computation center as an evolving public utility. Persons who take this view are interested in the accessibility of the center, the reliability of its operation, and the ease with which it is used. Others see the computation center as an advanced engineering firm producing complex technical products of limited lifetime. Persons who take this view expect us to regularly produce innovations of markedly increased value to the community. One who sees us as a utility thinks we should operate similarly to an electric company. One who sees us as an engineering firm would expect us to operate like an aerospace firm. Still others see us as a combination library and service facility. Persons who take this view generally have a particular puzzle to solve, and are directed to the computation center as the proper place for solving such puzzles. These persons expect us to operate like a hospital. We did not exclusively use any of these view points in planning our organization, but look upon ourselves as an ordinary industrial activity with a highly technical product line.

While the strong industrial organization does not preclude decentralized operation, it certainly prefers centralization. We have one large computing center with several smaller centers for specialized uses. These centers, as a matter of policy, cannot rent time nor compete with the main computing center. This involves no fundamental intellectual argument; we would actually prefer several strong centers to one strong center *if* there was enough capacity to run the largest program as well as the smallest. We have one center almost solely to insure that our facilities are fast and complex enough to satisfy the larger user. There is also the point that access is more uniform and controllable from one point than from several.

Many universities have confined their computation centers to one educational department, generally under the supervision of a chief scientist. Most often, this center is in the mathematics or electrical engineering departments. The most positive argument for this approach is that an umpire is needed to settle questions of priority and capacity, and that a single, strong voice will provide cohesive direction. This approach, however, simply does not recognize that the computer is now a fundamental tool required by some and of great utility to others. At Carnegie, we have a sufficient supply of technical talent and report to a high enough administrative level to facilitate, if not insure, reasonable growth and service. While we have broad participation and use at Carnegie, there is still an aggressive exploration of new applications in all areas, particularly in areas other than mathematics and electrical engineering.

A popular organizational approach is project management. In such an organization, no single individual is solely concerned with either operations or programming, but certain key individuals have responsibility for key projects within the center. Whether this is a good organizational plan depends upon the nature of the computation center. If, as at Carnegie,

there is an underlying fundamental operation to be carried on day after day; i.e., certain large programming service facilities to be maintained according to a set schedule of availability, then, in fact, the project leader concept is unsatisfactory. If, however, the computational facility is primarily used as a job shop with no fixed facilities other than the computer, then it is conceivable to organize project leaders under a chief engineer or, if the size of the installation warrants it, under a chief engineer who worries about technical problems and a project manager who worries about due dates and budgets.

Having discussed various other organizational schemes, we now turn our attention to that selected at Carnegie. We seek to provide the user with a stable, economical service of broad capability, which is sensitive to his needs, and which can easily be evaluated. On the other hand, we tend to have our own world view and to be inflexible in particulars. We will now discuss how we apply our philosophy considering the various administrative parts of the Center, how they are like their industrial counterparts, and how they contribute to the total.

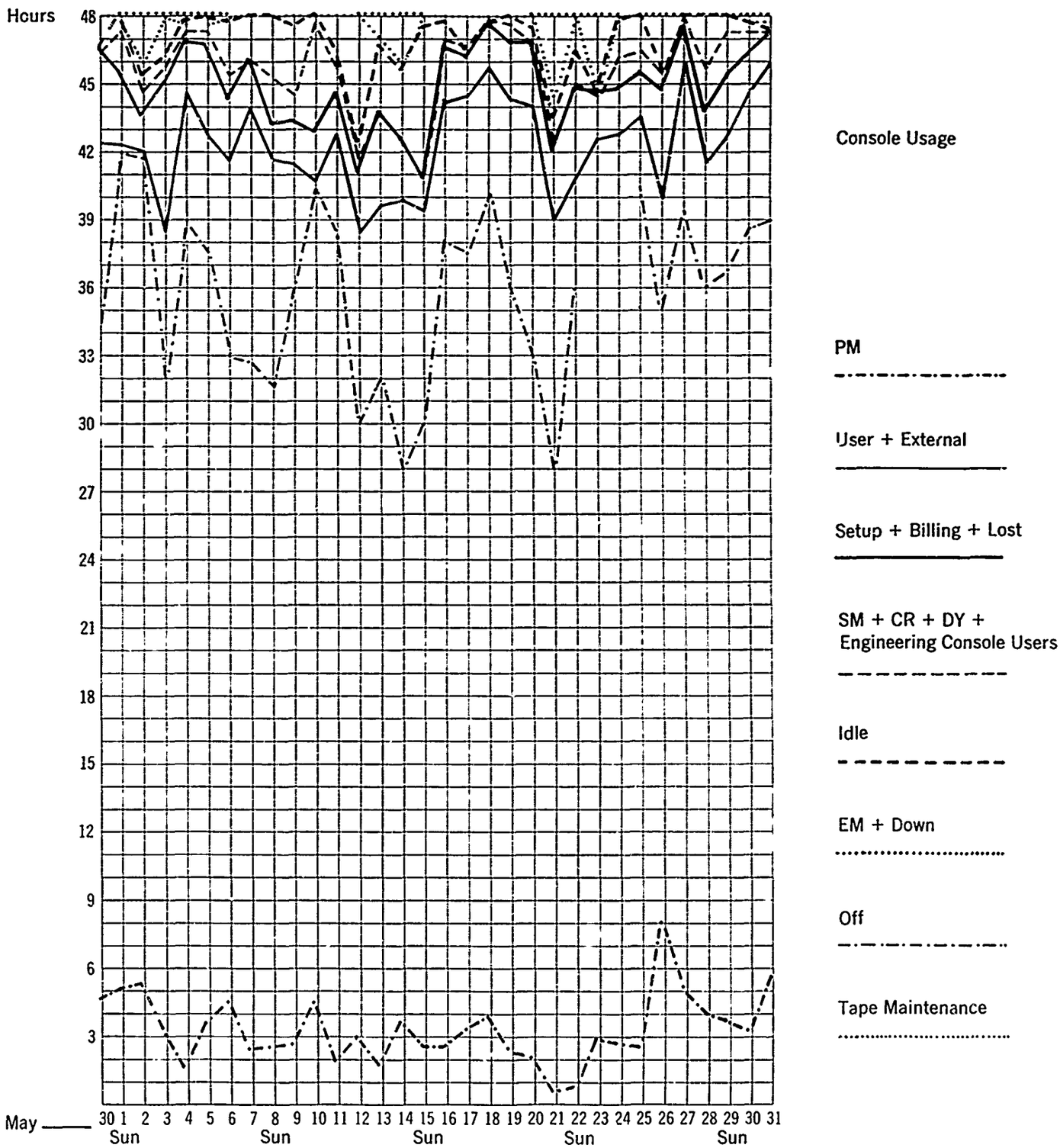
Fundamental to our point of view is the notion that, as a center, we produce one or more products which are of use in achieving our objectives. These objectives are stated in terms of interaction, if we can measure the pressure to interact and the amount of interaction taking place, then we can tell if we are succeeding as an activity.

Measurements come in pairs; if we produce a large quantity of low quality moments, we will have failed. It is equally true that a limited quantity of high quality moments would be, in our case, a failure. The organization we have parallels two underlying processes These processes are designed to produce both quantity and quality. We feel that we are 'business-like' first because we deliberately seek objective measures of performance, second, because we seek two such measures for every organizational activity (one measures quantity, the other measures quality) and, finally, because we seek to constantly improve our performance by redesigning the processes whose output we are measuring.

Fundamentally, of course, we produce 'moments' in time. These are produced inexorably as the day progresses. Good moments cannot be stockpiled, nor can bad moments be repaired. The quality of the moment can be judged in a significant way—it is judged by whether anyone is able or willing to consume it. Some moments are of routine significance and capability, others are of high cost and complexity, still others are useless to anyone.

The Manager of User Relations and Planning is charged with the responsibility of producing quality. He is to ensure, first of all, that the product offered will be consumed and, secondly, that distribution within the overall area is in accordance with the policy. This manager has three functions reporting to him: the User Consultant, who is charged with helping users utilize the product, and with interpreting the users' viewpoint; the Technical Writing Staff, which is charged with maintaining documentation at the highest level consistent with the technical information available, budgeted funds, and user interest; and the Planning section, which is responsible for preparing long-range and short-range plans. The long-range plans are two- and five-year projections of facilities to be made available, and the short-range plans are the yearly budget, schedules for facilities and software installations. A function that logically belongs here, but which is not at present included, is Systems Maintenance. This corresponds to the field engineering, or technical support, facility in an industrial concern. It belongs here because an organization's Production Department is notoriously insensitive to complaints.

While our Planning and User Relations Department is concerned with interpreting the Center and its policies to the users, and representing the user viewpoints to the Center, the Operations Department is charged with producing marketable moments. It does so by minimizing the number of moments which are not consumable. The Department is composed of three groups, each of which contributes to the achievement of its objectives. The first group is the Operators. They, in conjunction with maintenance engineers, control the


Hours
48
45
42
39
36
33
30
27
24
21
18
15
12
9
6
3
May
30 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Sun
Sun
Sun
Sun
Sun
Console Usage
PM
User + External
Setup + Billing + Lost
SM + CR + DY +
Engineering Console Users
Idle
EM + Down
Off
Tape Maintenance

activities of the hardware/software complex which is the capital equipment utilized in moment production. The second group, Systems Engineering, is responsible for recording the disposition of every moment of the day and determining the best possible allocation of our resources to meet current demands. Starting from a base number of hours, it allocates the time into various categories. An example of this analysis for May, 1966 (a heavy month in a university) is shown in Figure 1. The third group, Systems Maintenance, is charged with both routine and emergency maintenance for all released hardware and software of the Center.

In a volatile technical environment, a company which does not produce new products will soon lose its customers. There are splendid opportunities at Carnegie to exploit technical advances which simplify operations or increase utility. We, therefore, have a Product Design Department. Actually, it is organized as two separate efforts—Programming and Engineering.

Engineering serves the useful function of keeping the hardware used in the Center as current as economically feasible. The engineers are responsible for keeping us abreast of hardware technology and recommending changes to, or alterations of, the installed equipment. They also design special equipment as needed.

Programming is charged with designing the software facility which, together with the hardware facility, produces the product. This is rather like carrying coals to New Castle because the faculty and graduate students who are excellent programmers—even system programmers—outnumber the programming staff 10 to 1. Nonetheless, the mastering of any given field of programming is a full-time job in itself. Programming is organized into three groups: Systems Programming, which concerns itself with all long-term, non-language software development; Language Development, which is self-explanatory; and Special Products, which handles any development which is either not clearly in a specialty group or is too large or too small for the specialty groups.

The graduate department in computing science offers us a research environment of high quality. It also has the feature of consuming the basic product of the business. The output from this research group as developed by design and produced by operations is what keeps us on top of our market. It also gives us an independent, in fact aggressive, evaluation of our end product. Choosing projects to implement is our hardest single management task. That task would be impossible without the counterbalancing forces of marketing and production in our counsels.

The opportunities for weaknesses in the organization selected are many. This is easily verified by observing similar organizations in the industrial world. Production can become more concerned with stability than with being responsive to customer needs. Marketing can lead us into products which are too esoteric or too mundane. Marketing and production can become involved in struggles to dominate planning. However, the positive aspects of choosing this particular organization are first, it works; second, its mechanisms are, generally speaking, well understood by the administrative interface of the Center; and finally, much literature is available which proposes, outlines, and explains the utility of various administrative formalisms in such a context.

The development of a cohesive management structure is meaningless if one does not have the ability to correlate authority and responsibility. The ability to assign responsibility is in most concerns linked to the ability to measure results. Thus, a key part of any organizational question is the accounting theory used to judge it.

Project accounting seeks to collect the cost associated in achieving any clearly defined goal. In such a system, all expenses which can be clearly traced to a specific effort are collected and analyzed against funds budgeted for the effort. This method of accounting is used in many commercial computer centers. It is inherited from design projects of a similar nature, and also gains feasibility if one has a project type of organization.

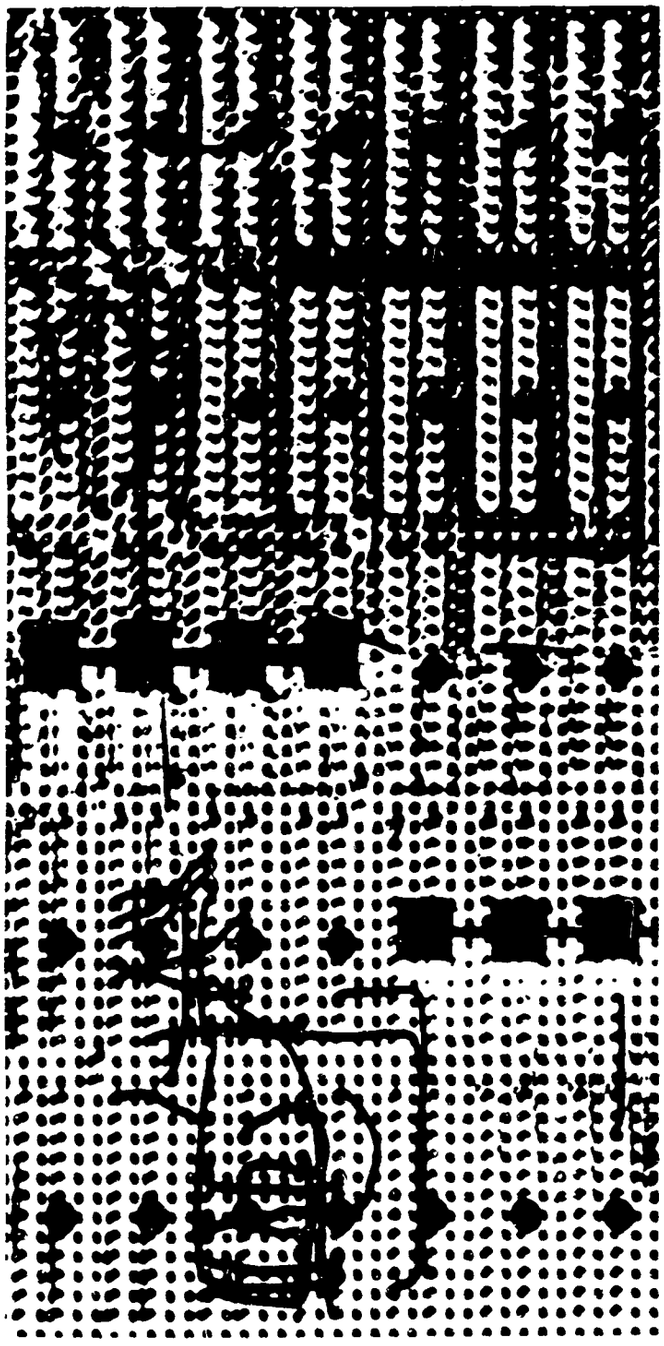
One way of spotting such an organization would be to seek for a situation in which a large percentage of compiling and assembly occurred. But should such a center ignore its compiling and assembly programs?

If, as at Carnegie, a major portion of the time is devoted to these tasks, then they are the *production* programs of that center. It is puzzling to see American business concentrating on efficient operation of its "production" programs but either not seeing or not caring about the efficiency or speed of assemblers and compilers. This is true even where these programs occupy 20 to 30% of computer time at an installation. Some measure of the magnitude of the difficulty can be sensed by realizing that the initial announced speed of the OS/360 assembler was approximately the same as the speed of Carnegie's ALGOL Compiler, despite the fact that the assembler was running on a machine basically many times faster. Other comparisons between COBOL or FORTRAN compilers have revealed order of magnitude differences of which most users were unaware. Another very similar situation concerns the debugging aids available at a center. At Carnegie, the addition of a helpful debugging concept in ALGOL—if it cut the amount of debugging in half, for example—might free 20% of machine capacity. One must conclude either that only in a shop which uses no formal systems at all would it be a good practice to collect cost only on a project basis or that to truly control all variable cost, the nature of a project must be stretched to include a host of quasi-permanent projects which are concerned with non-organizational facets of the environment.

The accounting theory used by many centers is that of product cost. There are several different versions of product costing. The one most commonly used in computation centers is full absorption cost accounting. That is to say, every dollar spent is eventually absorbed into the cost of some product. The standard products for control purposes are computer hours. This ties back to our earlier discussion of what our products are. Therefore, the control situation, as it exists now, is to take the total dollar expenditure and divide it by the total hours produced to arrive at an average cost per hour.

This practice leads, in the first place, to severe complications in discussions with auditors and accountants and, in the second place, to poor operational control of the center. Most accountants begin with the assumption that product costing is the only meaningful way to cost account a center. This leads to the derivation of a formula, as outlined above, and the charging of average cost to everyone independent of the demands they make upon the facilities. This means that a customer who uses ALGOL, developed at a high cost over a period of years, pays exactly the same price as one who uses a card punch utility routine which came with the machine. This is analogous to charging the same for an airplane and an automobile because both burn gasoline! It also means that a large research project, which makes heavy demands upon the system, pays exactly the same rate as an undergraduate who only uses the system when no one else needs it. Further, a computation center which has a mixture of users paying this rate penalizes itself if it improves its performance because speeding up its operations will cut its billing and force it to attract new usage. The seasonal nature of work at a university complicates this even more. We only add research customers in the fall. This subject is exhaustively treated in [2].

Yet another alternative is responsibility accounting. This approach traces expenditures to the person who actually controls the expenditure. Many computation centers of the author's acquaintance are organized under some variation of this principle. The difficulty is that most of them use this scheme to organize and use product costing to account, hence they are inherently incapable of reacting to the information from their accounting systems. Responsibility accounting is achieved by the division of the computation center into cost centers, each such center being in control of some specific aspect of the operation. Usually, the division is one of operations, design, and support activities. The first question is, "Does such a cost center structure adequately reflect the responsibility situation?" It does represent a good reflection of organizational control. The operational characteristics of the center are such that there is a short-range control loop (marketing, operations) which requires fast positive decisions and pragmatic oriented people to implement them. There is another longer range control loop (design) which requires complex decisions implemented by theory oriented individuals. The two functions do not comfortably report to the same person because they



require more filtering and balancing than can be accomplished in one management level.

The second question is, "Do the accounting principles employed tend to support or undercut desirable improvements in performance?" Whether responsibility accounting is adequate depends more on correct cost distribution than on anything else. If, for example, non-deliverable moments are not isolated and charged to someone, then there will be no incentive to eliminate them. At Carnegie, we isolate several categories of lost time; i.e., non-productive time, and charge these, figuratively speaking, to the Manager of Operations. Thus, he has an incentive to improve his performance. However, we have not taken the obvious step of isolating programming facilities and deriving a measure of their efficiency to judge the programming group. One might, for example, charge all compile time to programming. This is really no less objective than holding the Manager of Operations accountable for all computer time as a deliverable product. If compile time were charged to the programming group, and we could see that we spend almost 30% of our time compiling, we would feel the need for more programmers than operators. This whole issue needs a great deal of analysis and represents the next area for attention at Carnegie.

Responsibility accounting, however, will not satisfy the accountants unless it is coupled with a relevant product costing system. Some portion of assignable cost is fixed and not controllable. Heretofore, it has been assumed that these costs should be lumped and absorbed uniformly. Even Project MAC has indicated that it intends to continue this demonstratedly misleading practice in Multics by charging swap time to the user. It has been shown here that such a practice has ramifications which are far reaching. One of our prime projects during the coming year will be to construct a more reasonable marriage of responsibility accounting to product costing.

A computation center contains parallels with a modern industrial corporation. While much work remains to be done, this paper has sketched out a preliminary definition of the product of a computation center, and has described an administrative mechanism for its

production and distribution. Many situations are unique to the university environment. Chief among these are the great flexibility required, and the relative inadequacy of the accounting systems available.

Many academicians will not agree with our concept of business-like management. The chief differences with this point of view have been analyzed and our own point of view has been defended. Nevertheless, like any good business, we do not feel that we will be organized in the same manner three or four years hence.

In which direction should we change? How much should we change?—these are very important questions. We have presented in this paper a context in which these questions can be meaningfully discussed. Hopefully filling a gap in the literature on the management of computer centers, we have nevertheless left many areas untouched. It is hoped that they will emerge in future papers by ourselves or others.

References

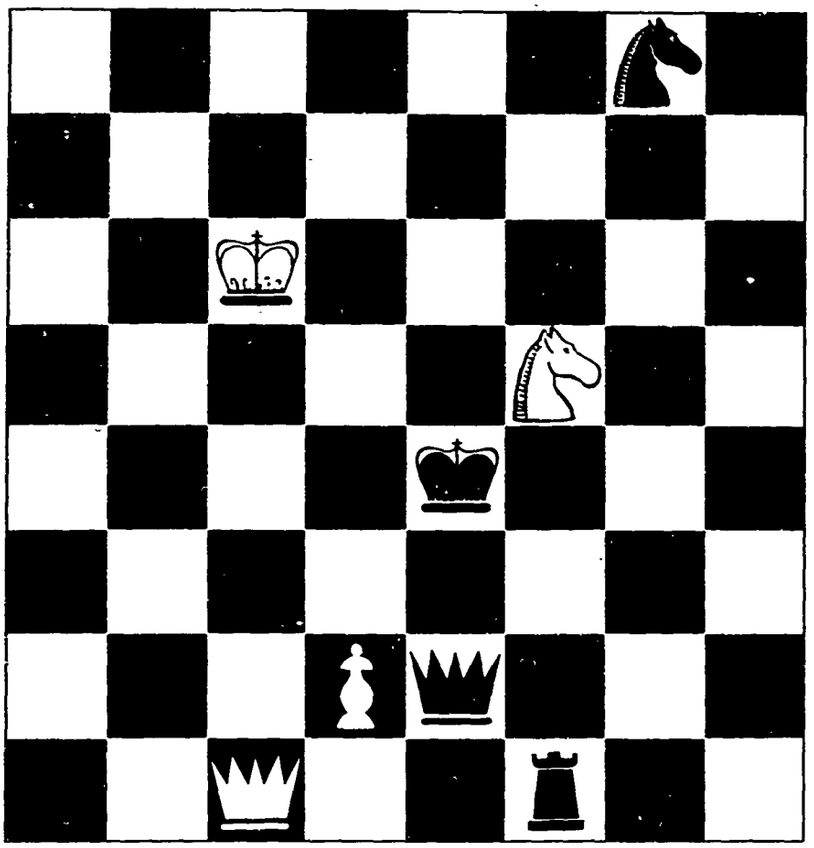
1. Newell, Allen, Perlis, Alan J., and Schatz, Edward R., "Proposal for a Center for the Study of Information Processing" submitted by Carnegie Institute of Technology to the Advanced Research Projects Agency of the Department of Defense, October, 1964.

2. National Academy of Sciences—National Research Council, "Digital Computer Needs in Universities and Colleges," 1966.

3. Sloan, Alfred P., Jr., *My Years with General Motors*, New York: Macfadden-Bartell Corporation, 1965.

4. Veblen, Thorstein, *The Theory of Business Enterprise*, New York: Charles Scribner's Sons, 1932.

Red



White

White Pawn (Alice) to play, and win in eleven moves.

As the chess-problem, given on a previous page, has puzzled some of my readers, it may be well to explain that it is correctly worked out, so far as the *moves* are concerned. The *alternation* of Red and White is perhaps not so strictly observed as it might be, and the "castling" of the three Queens is merely a way of saying that they entered the palace: but the "check" of the White King at move 6, the capture of the Red Knight at move 7, and the final "checkmate" of the Red King, will be found, by any one who will take the trouble to set the pieces and play the moves as directed, to be strictly in accordance with the laws of the game. Christmas, 1896

Lewis Carrol
Through the Looking Glass

Dr. Allen Newell, Institute Professor

On the Representations of Problems

Computer science is concerned with understanding the varieties of information processing. What processes will invert a matrix? Will parse a sentence? Will solve a set of simultaneous equations? Will play a game of chess? In each case an information processing system faces a task environment of which some things are known *a priori*, and other things are unknown. The former provides the basis for the system's initial structure The unknown things must be discovered by the system during the course of processing, or, at least, enough of them to permit determination of the final desired information. These acts of discovery, of course, must themselves be performed by the system, and hence be based on the initial knowledge that is available. If these acts of discovery are well enough contained, as in the assimilation of the parameters set for a subroutine, we may not even think of them as discovery. If, however, a variable number of successive acts of discovery are required, each building on prior ones, then discovery is a most appropriate term, and the system may be termed a problem solver. It is clear that if we want computers to become more and more sophisticated, we must come to understand such processes.

To date much of the knowledge that we have gained in developing programs that solve problems can be expressed in the following model of problem solving. The problem is presented to the problem solver in some external representation. The problem solver selects a *problem space* in which to work on the problem, and translates the problem into this space. The elements of a problem space consist of *states of knowledge* about the problem. One may think of these as data structures in some kind of language in the problem solver's memory. It must be possible to translate into this space both the *initial situation*—that is, the givens of the problem—and the final situation—that is, the

knowledge that will indicate that the problem has been solved. The problem space also contains a set of operators, which permit the problem solver to obtain new states of knowledge from old. Again, it must be possible to translate the transformations and manipulations that are permitted by the definition of the problem into the operators of the space.

Trying to solve a problem, once it is represented in a space, is a matter of *search*. The problem solver starts with what he knows, and applies the operators to get more information, and then again to get still more information. Thus, the scheme is just as sketched in the beginning: a succession of acts of discovery in an attempt to arrive at the desired state of knowledge where the solution is known. Intelligence enters into this search process by the evaluation of states of knowledge, the selection of operators to apply, and the decisions to start over when a dead end has been reached. The various rules used to attain selectivity are called heuristics, and this model of problem solving is consequently called *heuristic search*.

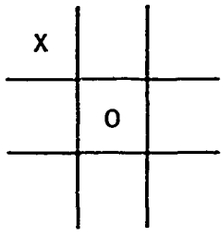
There is no need for the problem solver to limit himself to a single problem space; it is solely a means to solve the problem. He may abandon one and try another. He may switch back and forth between two; for example, a simplified model and a more complete one. Similarly, the problem space is not the whole of the problem solver. Other processes exist for selecting the problem space, for translating to and from the problem space, and for retrieving relevant information.

Without excessive oversimplification, it may be asserted that all the successes so far in problem solving programs have come from the investigator choosing a task, discovering a suitable problem space, and programming a computer to search for solutions in this problem space. The work on chess and checker programs, on theorem proving, on puzzle solving, and on a number of management science tasks all fit this pattern. The essential discoveries are of two kinds: the sufficiency of a problem space formulation for significant performance on the selected task; and the efficacy of various mechanisms of evaluation and selection. A more detailed description of this view with suitable qualifications and an extensive listing of heuristic mechanisms can be found in Newell and Ernst.[6] Here we wish to follow a different thread of the argument.

This view of problem solving suggests that one should be mightily concerned about where the problem space comes from. More generally, it is a common notion that hard problems are solved by finding new "viewpoints"; i.e., new problem spaces. In human problem solving different people use different problem spaces, especially in regard to the operators that are available.[5] Not surprisingly, those with objectively more powerful spaces, do better. One can argue, for instance, that the learning of Samuel's justly famous checker playing program[8] is inherently limited by the framework within which it plays; i.e., by its problem space. That no one yet has proposed a better space does not remove the concern.

In this essay we will reflect a bit on the problem of representations for problem solving. The topic has been a prominent one this spring at Carnegie. Owing to the presence of Dr. Saul Amarel (of RCA's Research Laboratories at Princeton) as a visiting faculty member, an advanced seminar was held to explore the topic. The area is certainly not ripe yet for systematic treatment, but we can proceed piecemeal, making use of several recent pieces of work at Carnegie to illustrate various aspects.

Does representation make any difference? Consider the four games whose rules are briefly sketched below. Each is played by two players, 1 and 2, moving alternately with 1 starting. If neither player wins, the game is declared a tie.

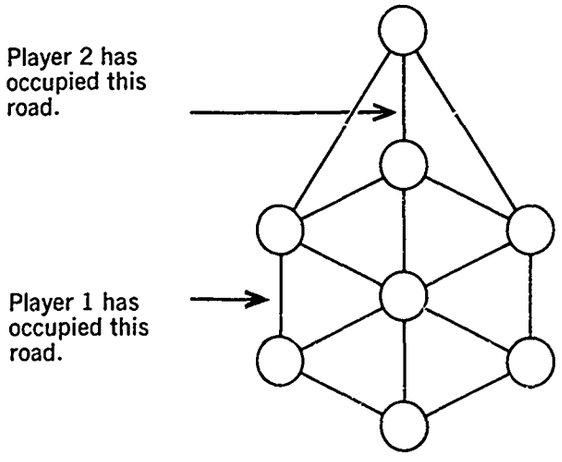


Tic-tac-toe: Played on a square board, as shown. At his turn, each player puts a characteristic mark in any unmarked square, X for player 1, O for player 2. The first player to mark an entire row, column or diagonal wins.

Pool: 1 3 4 6 7 8 9

Player 1: 2 Player 2: 5

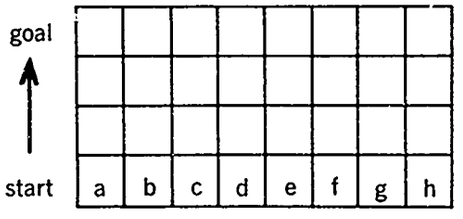
Number Scrabble: The nine digits, 1, 2, . . . , 9, are used to label a set of blocks, which constitute the initial pool. At his turn, each player draws a block from the pool. The first player able to make a set of three blocks that sum to 15 is the winner.



Jam*: Play takes place upon the network of roads, as shown below. At his turn each player can occupy a road (all of it) and thus jam (i.e., block) access to the town on the road. The first player who succeeds in isolating a town, in the sense of jamming all roads leading to the town wins.

*Developed by J. A. Michon.

	a	b	c	d	e	f	g	h
M1			x		x			
M2	x			x			x	
M3		x				x		
M4	x					x		x
M5		x			x		x	x
M6			x	x				x
M7		x		x				
M8			x			x	x	
M9	x			x				



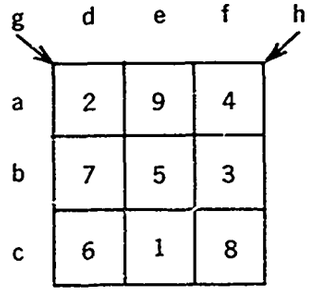
Race: Played on a rectangular board, as shown below. There are eight race horses in the starting row, and the first player to get a horse to the goal row wins. The players select from the same set of moves, but each move may be used only once. The effects of these moves on the horse is shown in the matrix. An X means the move affects the horse; a blank means it doesn't. If there is an X, one of the three things occurs:

if the horse belongs to the moving player, then it is advanced one toward the goal;

if the horse belongs to the opposing player, then it is disqualified from the race;

if the horse doesn't belong to either player, then it becomes part of the moving player's stable, and is advanced one.

All four of these games are isomorphic to each other; that is, they are all tic-tac-toe. Number Scrabble comes about because of the magic square, as given below, so that all sums of columns, rows and diagonals adds to fifteen. Jam is simply the projective dual, where lines are taken as points and vice versa. These are labeled on the magic square. Race comes about by abstracting the eight goal states, and relating the nine possible moves directly to the effect on these goal states.



Does it make any difference which representation a player uses? Can he play Jam just as well as Tic-tac-toe? One thing is clear. Since the games are isomorphic, the game trees are the same: hence if the problem solver plays by exhaustive tree search the only difference can be one in the speed of processing. Further, any rule, either algorithm or heuristic, that can be expressed in one game must have its counterpart in the others. However, if we consider how easy it may be to discover these heuristics, or to implement them, there can exist differences. Actually, an analysis by H. A. Simon[9] shows that the differences between these representations is at the level of perception and memory. More complex problems than Tic-tac-toe sem to be required to show the ways in which representations affect reasoning (as opposed to perception). However, the reader may want to ponder these four versions (and other?) a bit.

An additional viewpoint is possible. Suppose we had a problem solver who, granting it somewhat greater sophistication than current game players, starts with the rules of a game and builds up for itself a set of methods for play. Then, depending on the version given it, our problem solver would generalize very differently to other games. For instance, the natural generalizations from Tic-tac-toe are Qubic (three dimensional Tic-tac-toe played in a 4x4x4 block) and Five-in-a-row, played on an infinite plane. But these are not at all natural from the viewpoint of Number Scrabble. Here it is natural to think of games defined by a set of numerically labeled blocks, with the sum and number of blocks to a win given. These, it appears, bear no relationship to the in-a-row games on a plane board. That there is a single point of inter-section depends on the special properties of magic squares. Similarly, the game Race lends itself to generalizing the kinds of moves possible—some moving two ahead, some backing horses up, etc., as well as boards with obstacles in the way. This suggests, perhaps, that the existence of good representations may depend on very special properties—e.g., on the "inter-section" of fundamentally different classes of situations—and thus that any general theory of them will be very hard to come by.

Representations for difficult combinatorial problems. Imagine a long line of identical soldiers. At some moment, call it t=0, the one at the right end, called the general, gives the signal to fire. All the soldiers (including the general) are then to fire simultaneously. However, each soldier can only communicate with the man to either side; so that, in fact, when the general gives the signal only he and the first soldier to his left can know it. We assume that an act of communication takes one time unit. The problem, then, is to specify the communication strategy of the soldiers. This has come to be known as the Firing Squad Synchronization Problem,[4] and was posed originally by John Myhill.

The problem is really one in the design of abstract sequential machines. Each of the soldiers is a finite state machine—that is, can be in one of k states, S_1, S_2, ... S_k at each instant of time, t=0,1,2, ... at each time, t, the machine jumps to a new state, which can depend on its own state, the state of the soldier to its left and the state of the soldier to its right. A final condition also holds: The complexity of the soldiers, as

measured by the number of states, k, is arbitrary, but fixed. However, the scheme must work for any length of line, n. In particular, n can be much larger than k. Thus, the obvious strategy won't work. This is to have the soldiers "count off" (as they say in the military), thus each labeling himself by 1, his position in the list. Then when the return signal comes back from the far end of the line, he knows to fire i time units later (when the signal finally gets back to the general). This solution takes only 2n-2 time units. This is certainly the minimum time possible, since one cannot solve the problem without at least some effect traveling all the way down to the end and back. However as we said, it is impossible, because the soldiers, having only k states, simply cannot count up to a number greater than k.

Thus, the problem is a real one. The first solutions were provided by John McCarthy and Marvin Minsky, then both at M.I.T. (Once you know it has a solution it isn't too hard to find at least one.) Interest then centers on how long it takes to get synchronized. The easy solutions take essentially 3n. However, E. Goto, of Tokyo University, showed that a minimal time solution does exist—that is, that the firing could take place in 2n-2 time units. His solution, which was only sketched apparently, took many thousands of states. Finally, Robert Balzer[2], at Carnegie discovered a minimal time solution with only 8 states,* and proved that this was indeed a solution for all n. However, he has not shown that 8 states are necessary.

Two questions now arise. First, why should all these people want to worry about this abstract problem; and second, why should I want to describe it here in a discussion of representation. As to the first, there is the faith, common through all of mathematics, that the deep study of hard and elegant problems gives rise to techniques that eventually find their way to broader application. There need be little concern over what qualifies a problem for this status since a form of natural selection applies. Easy problems do not survive, except as exercises in textbooks. Likewise, inelegant problems, when not required for practical purposes, do not survive for free men will not work on them. The elegant ones get names (e.g., the Four Color Problem, the Traveling Salesman Problem), and accumulate a trail of researches in their behalf. It is, of course, too early to tell about the problem at hand. It has a certain appeal, and it has both a name and a small trail to date.

To see what the problem has to do with representation, we must step behind the scenes to see how Balzer proceeded. First, some representation of the set of machines is needed. Various standard ones exist, for the study of finite state machines is well advanced. A somewhat specialized variant is convenient for this problem. Let the states be indicated by capital letters, A, ..., Z. The schema of the form

XYZ→W

says that if a soldier is in state Y and if the man on his left is in state X and the man on his right in state Z, then the soldier will jump into state W. This form (called a production) gives an elementary component of a machine, and a large collection of them completely describes the behavior. For 8 states there will be 8x8x8 (=512) productions, one for each possible situation that might arise. Note that for a given triple, XYZ, there cannot be more than one production, or the behavior of the machine would not be well specified.

Now the design problem can be restated. Let the quiescent state be Q, the initial state of the general be G, and the firing state be F. Introduce also an extra state, X, to border the ends of the lines so that all soldiers have a man on both sides. For a given length, n, we can represent the conditions on the behavior of the machine for minimal time performance by a graph (shown for n=5).

	XQQQQGX	XQQ → Q	n=5
t=1	XQQQ X	QQQ → Q	
2	XQQ X	QQX → Q	
3	XQ X		
4	X X	No F occurs inside	
5	X X		
6	X X		
7	X X		
8	XFFFFFX		

Three productions are already given, since the quiescent state maintains itself in absence of an external signal. Thus, we can fill in the Q's on the left for early t, as

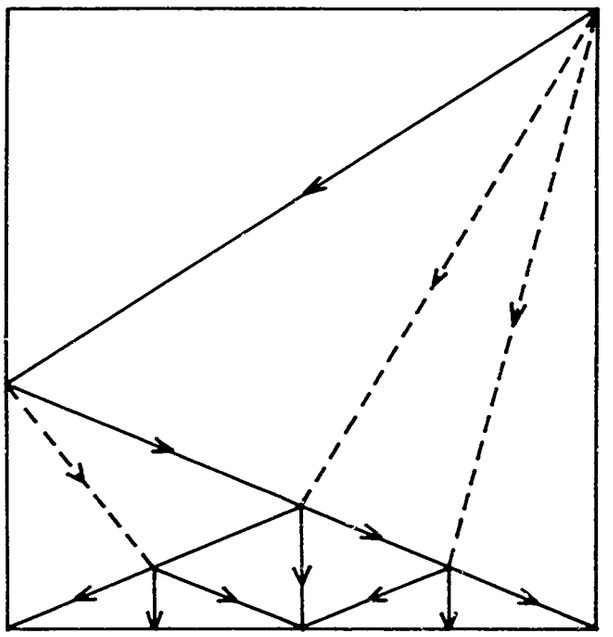
*Independently, J. Waksman provided a 15 state minimal time solution.[10]

they are already implied by these productions. We want, then, a set of productions such that it contains the three above, and the behavior of the resulting machine fits the conditions for all n corresponding to those above.

Posed this way we can easily translate the problem into a problem space. The elements of this space are partially specified machines—i.e., sets of productions. The operators are the acts of specifying an additional production. The initial element contains just the three productions above; the final one contains enough productions so no new ones are needed to define the behavior of the machine. Considering the 8 state case, the full 512 is an upper bound to this, but we may get by with many less. However, the number of machines we might have to consider is very large, since each of the 512 productions can transform to any of the 8 states.

Our design problem now looks very much like other tasks, such as chess, theorem proving, and so on, that have been tackled by problem solving programs. Balzer constructed several different programs to try to perform the design of such machines automatically, making use of several heuristics to cut the search to reasonable size. He was able by these means to show that no 4 state machine could solve the problem and to find several variants on the 8 state machine. However, that was the best he could do, and even with the 8 state case he had to prime the system with much that appeared necessary to him from the one 8 state system he had discovered by hand.

But clearly Balzer himself had found a solution, and manifestly not by working in the same problem space that the machine did. Therefore he had a different representation that was more powerful. Thus, the problem for us is to discover his alternative representation, and to see if it can be given to the machine. We do know some things about it. Balzer, and many other humans, think of the problem initially in terms of signals propagating down the line—with reflections, velocities, intersections, transparencies, etc. None of these terms is in the original formulation, where the concepts are only states and transitions. Thus, a diagram, like that below, can be used to rapidly convey the nature of the solution:



Furthermore, Balzer described the states in terms of the *functions* they performed. A state can be a *carrier* of the left traveling signal; or a *medium* for the left traveling signal. It can be a *residence* state (once established it never changes until F); or a *middle* state. For instance, one reasons as follows. The problem with the F state is that one cannot jump into it until the last minute; what one needs is a state that can cumulate—so that some men can get into it before others do, but such that not until all get into it will they all go to F. Let us call this state Z, the *ready-to-fire* state. Then one can write productions:

XZZ → F
ZZZ → F
ZZX → F

Fire when everybody is ready. These will be the only productions leading to F.

Further, the simple design choice is to make Z also a residence state, so that when a man becomes a Z, he will never change:

xZy → Z for x, y not Z

(We have used x and y as variables for other states.)

But this means we can add a permanent restriction to the set of conditions we are working toward, since the entire line at t=2n-1 must consist of Z's.

A whole collection of productions has been specified by adopting a single decision. Hence, in a problem space whose operators are design decisions, search would be much more efficient. In fact, Balzer incorporated these ideas, up to a point. Some functions are local; that is, their properties can be given solely in terms of classes of productions. It was possible to give the programs such local constraints, and it would select within these. By this means the search for alternative variants of 8 state machines was shaped to look in the right part of the space. However, some of the most important constraints are non-local; most notably, the function of middle states. This is a residence man that divides the region between two boundaries in half, and forms a further boundary that permits independent processing to occur simultaneously on both sides. No way was found to specify these to the program operationally.

There the matter stands at the moment. We have an example of a problem with one well defined space clearly inadequate for discovery, and the hint of another more potent one. If we could formalize this latter, perhaps we could get a program to solve the remainder of the problem to determine the minimum of states for the minimum time machine. The task is not just one of inventing a good discovery space, however, since one must verify that no possible machine exists (as defined in the more detailed space) that can do the job. The problem seems a nice challenge to those interested in how one discovers new representations.

An example of three spaces. Saul Amarel has provided a very pretty case study of a task in which one is able to move through three successive representations. We will only be able to sketch the matter, since considerable formal apparatus is necessary. The original monograph should be consulted for details.[1]

We start with the task of proving theorems in the propositional calculus, a form of symbolic logic that has received much attention from researchers trying to build problem solving programs. The external representation—how the task comes to be expressed in the first place—is taken according to a scheme called "natural deduction." This was developed by the logician Fitch and others as a formalization of logic that seemed to correspond to the way humans reasoned. Its flavor can be indicated by the following stylized argument, where I am interested inproving $(p \supset q) \supset (-q \supset -p)$ (in words "p implies q implies that not−q implies not−p).

To prove $(p \supset q) \supset (-q \supset -p)$ show that if suppose p q then $-q$ $-p$ follows.

Hence, suppose $p \supset q$.

To prove $-q \supset -p$ show that if suppose $-q$ then $-p$ follows.

Hence, suppose $-q$.

To prove $-p$ show that if suppose p then a contradiction follows.

Hence, suppose p.

To prove a contradiction prove q and also prove $-q$.

To prove q prove p and also prove $p \supset q$.

Prove p because p supposed.

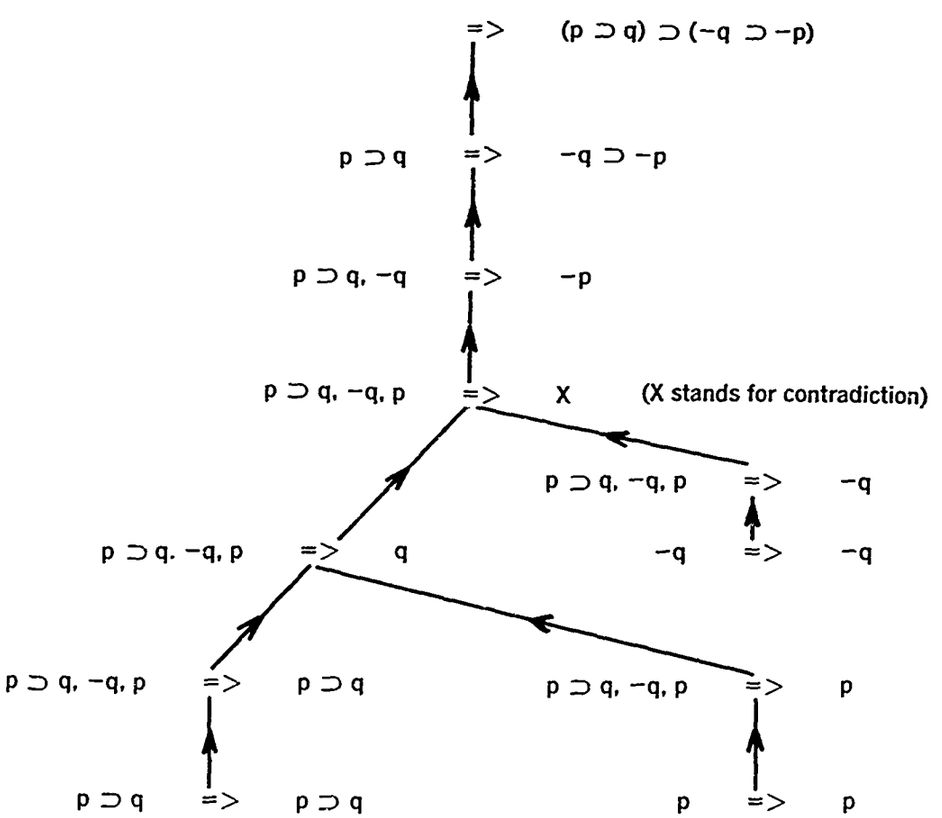
Prove $p \supset q$ because $p \supset q$ supposed.

Prove $-q$ because $-q$ supposed. QED

As you can see from the style of argumentation, this scheme is also called the method of suppositions.

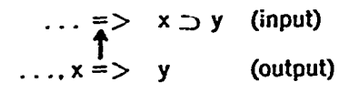
First, this task must be translated into a problem space. A natural one, close both to the system as given by logicians and to other attempts to construct problem spaces for theorem proving, is to take the steps in the argument as the elements in the space and the rules of inference as the operators. These rules of inference are the generalized versions of the arguments used above; e.g., "to prove $x \supset y$, suppose x and prove that y follows." The initial step is simply "prove x," where x is some expression in the language. However, all later steps have various suppositions associated with them. Consequently the element of the space must give both suppositions and the expression to prove. Using => to separate the suppositions from the goal expression,

we can transcribe the argument as a tree:



Terminations of the tree occur when the goal expression also appears as a supposition; we have indicated these by rewriting the step with only the critical expression on the supposition side. The proof is a tree because two independent things sometimes require proof to establish the desired result: in one case, q and −q to get a contradiction; in the other, p and p ⊃ q to get q.

In this first problem space the operators (16 in all) work backwards.



The three dots indicate that whatever other suppositions already exist carry over. Thus, the arrow represents the direction of proof, not the relation from input to output.

Given appropriate representation inside the computer of the steps and the operators, we can search out from the given proof problem (the top one) trying to find a tree such that all branches can be terminated by a state of type x=>x, which is the only terminating (conclusive) state. At each stage several of the inference rules will apply, and so the tree of search will branch. This first space closely resembles other examples of heuristic search, and is a natural first way to formalize natural deduction.

The second space rests upon four observations, all of which are of quite general applicability, although they find particular, and successful, solutions in the problem at hand. First, one would like to find a subclass of elements in the problem space where the problem could be decided by simple means. This class should have some simple characterization so that the search can home in on it. In the present case a good candidate is the class of elements that only have atomic expressions; i.e., either x or ~x, where x is a letter. Then the theorem is proved if either the goal expression occurs as a supposition, or if some letter and its negative occur as suppositions; otherwise, the theorem is not valid. Thus, the goal is to get elements of this form, from which the decision will follow immediately.

The second observation is that one would like some recognizable set of features that distinguish a given element from being an element of the decidable class, and which can be eliminated one by one so that inexorably the initial problem can be transformed into a member of the decidable class. In general, of course, no such set of features need exist. However, in logic the connectives, for example, provide such a set. The occurrence of any connective indicates a compound expression which needs to be broken into its atomic constituents. Furthermore, there always exist sequences of operations that will do this without introducing new connectives. Our example shows this process of elimination clearly. Sometimes the application of several operators are necessary. The attempt to eliminate these features gives a sense of direction to the search for the proof. Even though the element is a long way from a member of the decidable set, one knows pretty much what to do next—eliminate connectives.

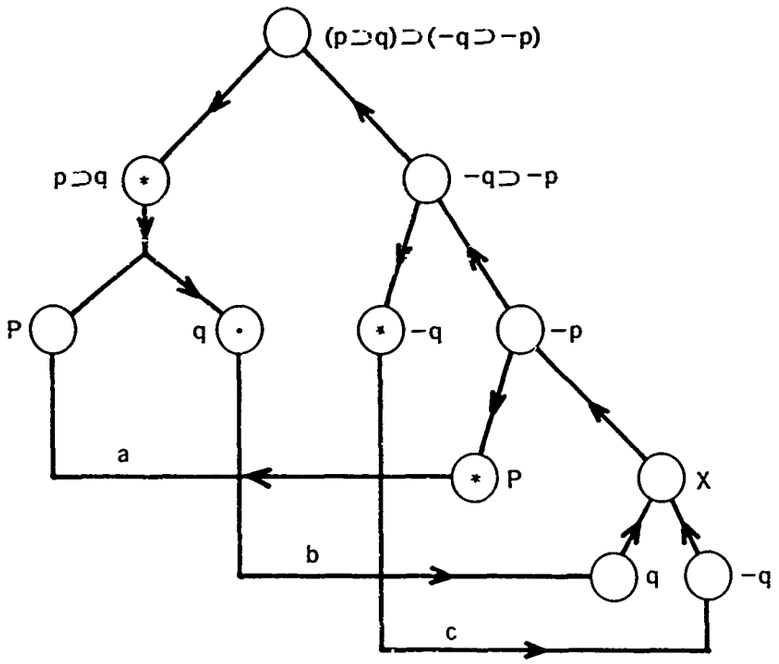
The third observation is that if characteristic sequences of elementary operators get applied, these may be bundled up into single large operators that express the final outcome as a function of the initial input. These Amarel calls *macromoves*. They cut the spread of the search tree down tremendously, since they replace little branching networks with long linear sequences; or, more precisely, with a single long leap. Again, in the logic task at hand such macromoves can be created for each of the elimination tasks. Different macromoves exist for each type of connective, and whether it occurs on the supposition or the goal side of the element. Each macromove eliminates one occurrence of a connective.

The fourth and final observation is that irrelevant detail may be possible. In the logic task this is again the case, and one can eliminate logically redundant expressions from the supposition. As an extreme case, if two suppositions were identical—e.g., x⊃y, x⊃y=>z—one could be dropped. This would avoid the needless elimination of its connectives, one by one, by the macromoves.

Put all these features together and you get Amarel's second problem space. The elements are those elements of the first space that have no redundant expressions; the operators are the macromoves that eliminate connectives. The search is now an inexorable downhill slide toward a decision. However, questions of efficiency still remain, since the order in which operators are applied affects how soon a validation or refutation will show up. This problem space is much more efficient than the original. It is closely related to the formulation that Wang[11] used for his theorem prover for the propositional calculus. Here, however, we see this problem space in relation to the more primitive

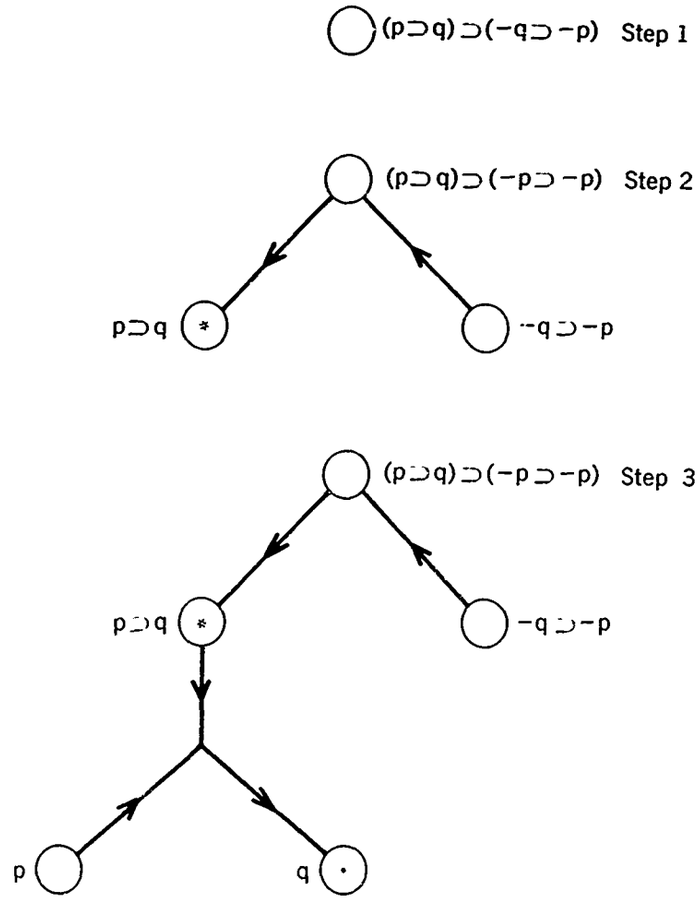
one from which it was derived.

Currently, the third space is best exhibited rather than described. We show below the proof of our example theorem in this third representation.



It is a connected graph; with each node representing an expression, as shown alongside. There are two kinds of nodes: source nodes (* and •), which correspond to suppositions or deduced formulas and destination modes (O), which correspond to goals. The condition for a proof is a graph all of whose destination nodes are contained in closed circuits, where the direction of the arrow counts. There are three such circuits in the example, associated with the links labeled a, b, and c. Each of these corresponds to one of the terminating steps in the original tree. The placing of these links "closed an argument" to use Amarel's descriptive phrase. Thus, the task of proving the example theorem can be viewed as the making of three arguments, corresponding to the three circuits.

A graph—it could be a single node or a fully connected proof graph—is the element of the third space; it is grown from the initial element, which is just the single destination node at the top, according to a set of elementary operators, which permits the graph to be augmented. These are related in certain simple ways to the operators of the first and of the second spaces. We reproduce the first steps below.



We called these operators "elementary," because the graphical representation permits the goal of proof construction to be rephrased as one of closing circuits. Amarel has written a set of procedures that prescribe how to apply these elementary operators so as to construct circuits. These procedures, which are the real operators of the third space, take into account the structure of the total graph to not only close circuits that can be immediately closed, but to prepare the way for the circuits that must eventually exist when the graph is full grown. These can be foreseen from the existing destination nodes, which one knows must eventually lie in circuits.

There are numerous features of this third space that have not been fully explored yet. The directed line segments are like flows, but flows of what? information? dependency? Notice also that the top part of the final graph closely corresponds to the tree representation of the expression to be proven. Is this the harbinger of yet another transformation of the problem which starts with a structural description of the expression that can be written down immediately? Although it is possible to describe in some detail the correspondence of features of this third space, it is not yet possible to describe its genesis from a set of general considerations that could be applied to the second space to yield the third space as an output. Finally, humans find the third representation effective because of its spatial characteristics—they can detect various global properties of the whole graph "at a glance." Amarel's operators capture some of these clues, but by no means all.

Many tasks, one representation. So far we have taken the viewpoint of a given task and how it can be represented in a suitable problem space. From the other direction, the question becomes how many tasks can be effectively represented by a given problem space. This latter is one way of posing the problem of generality; of how to get a single problem solver to solve many problems. For if the number of problems is very large, while the resources and preparation of the problem solver is limited, surely the same resources must be used over and over again.

Some research by George Ernst[3], just completed this year, helps to formulate this viewpoint more clearly. Ernst worked with a problem solving program called GPS, which has been in existence for several years[7]. Although the program has had pretensions of generality, being organized with a clear separation between problem solving processes and task environment processes, it had only been used on three different tasks when Ernst took over. The program is a heuristic search program. It builds goals to try to solve problems in a space consisting of situations and operators that work on these situations. Its specialty (if it can be called that) is the use of means-ends analysis as the main search technique. This involves characterizing the difference between the current situation and the one desired, and using this characterization to select the operator to apply next. In addition GPS can set up subgoals other than the main one. These come mostly from the desire to apply a given operator when the conditions for application are not fulfilled in the current situation.

For GPS to work on a task it is necessary not only to build a correspondence between the elements of the externally given task and the operators and objects of the formal problem space in which GPS works, but also to make it possible to perform all of GPS's problem solving methods. One can list these demands that GPS makes; there are seven of them:

Object comparison
Object difference
Operator application
Operator difference
Desirability selection
Feasibility selection
Canonization

The first and third are essential to any problem space: *object comparison* to detect when the problem is solved; *operator application* to get new objects. The others are peculiar to GPS's methods. The two difference demands relate to characterizing difficulties. The two selection demands obtain operators. The demand for canonization implies the ability to introduce and maintain a single name for common objects. Some of GPS's activities do not show up on this list, such as goal building and interpreting. These are already formulated and coded in a way that makes them independent of the particular task being worked.

If one were to retreat to the two basic demands (first and third), it would be easy to build a general problem solver. It wouldn't be very good though. It one were to add many additional demands, it might not be possible to find any way to get the information into the problem space, such that uniform processes could extract it. Holding the demands fixed implied that, as Ernst considered different tasks (eleven in all, by the time he was through), he had to extend and augment the internal processes of GPS in order to meet all these demands. These additions did not increase GPS's problem solving power, but only maintained it in the face of increased generality.

Let me give an example or two. In getting GPS to integrate symbolically, that is, to get from

$$\int te^{t^2}\,dt \quad \text{to} \quad \frac{1}{2}e^{t^2}$$

it was not possible in the initial GPS to express the final goal. One natural expression is, to "Get an expression that contains no more integrals to be performed"; that is, one in which "$\int$" does not appear. But the original GPS only permitted definite expressions (such as those above), and these are not sufficiently general. One immediately thinks of simply adding a program test for this final condition. Unfortunately, although this satisfies *object comparison*, it does not satisfy *object difference*. Thus, Ernst had to develop an expanded capability for expressing objects by descriptions, such that GPS could still obtain differences.

As a second example, the initial GPS permitted only operators that could be expressed as the transform of an input form to an output form. Such transformations are common in mathematical calculi:

$A^2-B^2 \Rightarrow (A-B)\cdot(A+B)$	algebra
$\int Fdx+\int Gdx \Rightarrow \int (F+G)dx$	integration
$P \supset Q \Rightarrow \neg P \vee Q$	logic

However, form operators are an awkward way to express many tasks, especially those that involve moving objects around. For instance, consider the missionaries and cannibals problem. The situations consist of three missionaries, three cannibals, and a boat, distributed in various combinations on the banks of a river. Initially they are all on one side; it is desired to get them all to the other. A natural way to express the operators* is

> The boat holds one or two of the missionaries or cannibals (all of whom can row), and may travel across the river from either bank.

Somewhat more formally:

Let x $\{0,1,2\}$
Let y $\{0,1,2\}$

> Move x missionaries, y cannibals and the boat from one side to the opposite side, if $1 \leq x+y \leq 2$.

> (It is implicit in "move" that the objects moved must exist on the side from which the move is made.)

To permit GPS to understand a language of move-operators, several demands had to be met. Not only must GPS be able to apply such operators, which is the obvious requirement, but it must be able to describe inabilities to apply operators in terms of a difference *(operator difference)*. In the initial GPS, where operators were composed of pairs of objects, this demand was met by the same process that provided *object differences*. The new language required a new process. In addition, both *desirability selection* and *feasibility selection* required new processes. In the initial GPS there was an explicit table of connections which tied differences to the operators that were relevant to their reduction *(desirability selection)*. But in the move operators variants of a set of operators may be expressed by variables inserted in a single expression. In the missionaries and cannibals example there is only a single operator, with variables for the number of missionaries, the number of cannibals and the direction of travel. Thus, no table of connections can select out the desirable subvariant. In adding move operators to the repertoire of GPS, Ernst had to invent ways to specify these variables to select the desirable operators, as well as provide processes simply to apply the operator.

This essay is not a place to cover all the modifications made; the original study can be consulted. From our viewpoint this work provides a step in understanding what is required of a representation that it serve for many tasks.

Four examples have been given, each in the nature of a case study, each somewhat fragmented. Some have attempted to provoke thought in the reader. Some have taken studies aimed in other directions and bent them to our purpose. Several places, where precision and technical detail were appropriate, have remained superficial. Still, the aim has been to pose a hitherto ill defined problem, and not to settle it. How one represents problems so that preconceived techniques—and that is all that the problem solver can have at the start—can work on them is of major importance in computer science.

*We ignore in this discussion the constraint that there shall never be more cannibals than missionaries on either bank, to keep the missionaries safe. It can be incorporated either in the operator or as a separate test.

References

1. Amarel, S., "An Approach to Heuristic Problem Solving and Theorem Proving in the Propositional Calculus," R.C.A. Laboratories and Carnegie Institute of Technology, (1966).

2. Balzer, R., "Studies in the Firing Squad Synchronization Problems," Unpublished Ph.D. thesis, Carnegie Institute of Technology, (1966).

3. Ernst, G. and A. Newell, "GPS and Generality," Carneige Institute of Technology, (1966).

4. Moore, E. H. (ed.) "Sequential Machines: Selected Papers," Addison Wesley, (1964).

5. Newell, A., "The Study of Human Problem Solving Protocols," Carnegie Institute of Technology, (1966).

6. Newell, A. and G. Ernst, "The Search for Generality," in W. Kalenich (ed.) Proceedings of IFIP Congress 65, Spartan, *1*, 17-24 (1965).

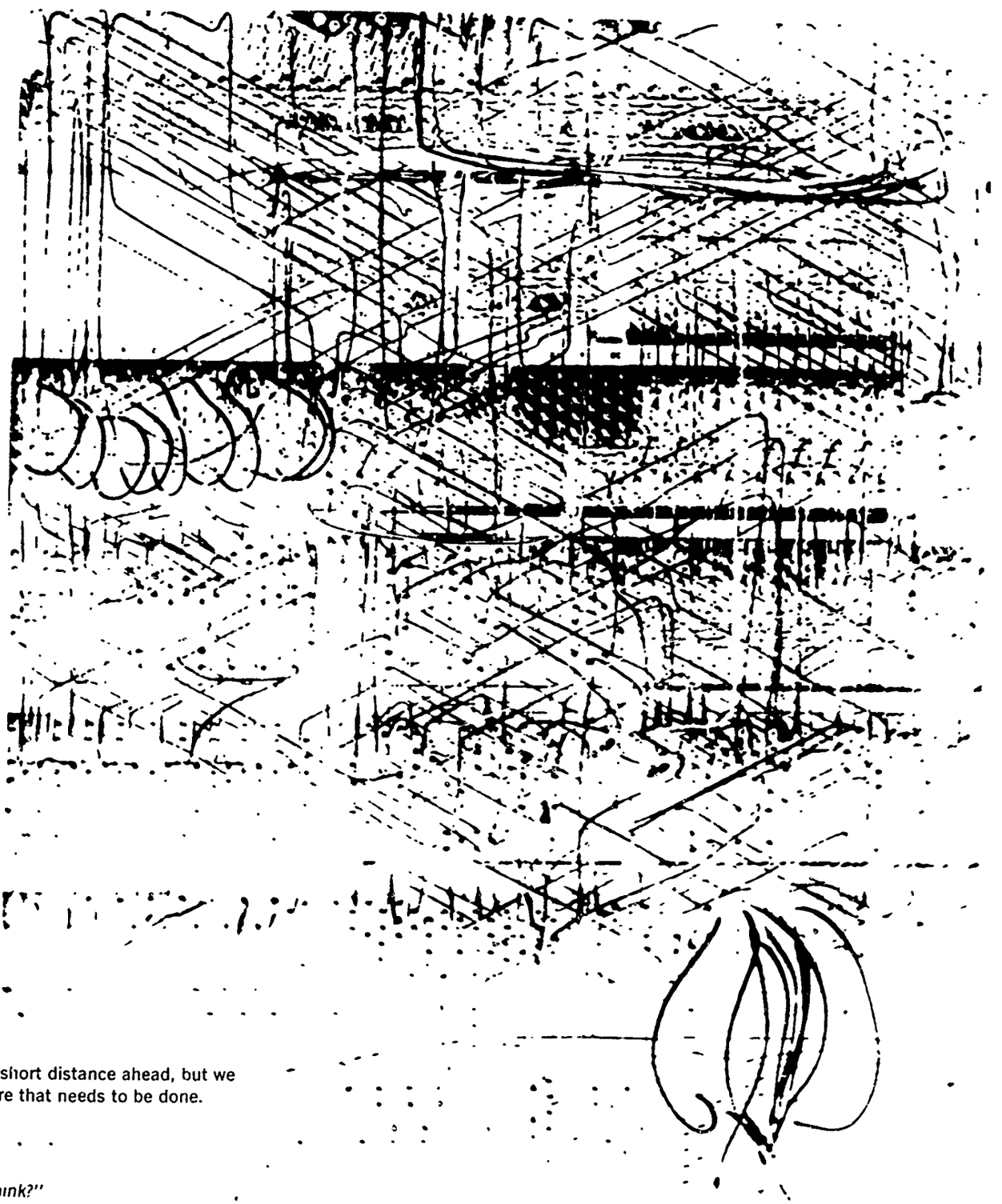
7. Newell, A., J. C. Shaw and H. A. Simon, "A Report on a General Problem Solver," Information Processing, UNESCO, Paris, (1959).

8. Samuel, A., "Machine Learning Using the Game of Checkers," IBM Journal of Research and Development, *3*, 211-229 (1959).

9. Simon, H. A., "Representations in Tic-Tac-Toe," C.I.P. Working Paper #90, Carnegie Institute of Technology, (1966).

10. Waksman, A., "An Optimum Solution to the Firing Squad Synchronization Problem," Information and Control, *9*, 1, 66-78 (1966).

11. Wang, H., "Toward Mechanical Mathematics," IBM Journal of Research and Development, *4*, 1, 2-22 (1960).



We can only see a short distance ahead, but we can see plenty there that needs to be done.

A. M. Turing
"Can Machines Think?"

Dr. A. J. Perlis, Head, Department of Computer Science

The Synthesis of Algorithmic Systems[1]

On what does and will the fame of Turing rest? That he proved a theorem showing that for a general computing device—later dubbed a "Turing machine"—there existed functions which it could not compute? I doubt it. More likely it rests on the model he invented and employed: his formal mechanism. This model has captured the imagination and mobilized the thoughts of a generation of scientists. It has provided a base for arguments leading to theories. His model has proved so useful that its generated activity has been distributed not only in mathematics, but through several technologies as well. The arguments employed have not always been formal, and the consequent creations not all abstract. Indeed a most fruitful consequence of the Turing machine has been with the creation, study, and computation of functions which are computable, i.e., in computer programming. This is not surprising since computers can compute so much more than we yet know how to specify.

I am sure that all here will agree that this model has been enormously valuable. History will forgive me for not devoting any attention in this lecture to the effect which Turing had on the development of the general purpose digital computer, which has further accelerated our involvement with the theory and practice of computation.

Since the appearance of Turing's model there have, of course, been others which have concerned and benefitted us in computing. I think, however, that only one has had an effect like Turing's: the formal mechanism called ALGOL. Many will immediately disagree, pointing out that too few of us have understood it or used it. While such has, unhappily, been the case, it is not the point. The impulse given by ALGOL to the development of research in computer science is relevant while the number of adherents is not. ALGOL has mobilized our thoughts and has provided us a base for our arguments.

I have long puzzled over why ALGOL has been such a useful model in our field. Perhaps some of the reasons are (a) its international sponsorship, (b) the clarity of description in print of its syntax, (c) the natural way it combines important programmatic features of assembly and sub-routine programming, (d) it is naturally decomposable so that one may suggest and define rather extensive modifications to parts of the language without destroying its impressive harmony of structure and notation, (There is an appreciated substance to the phrase "ALGOL-like" which is often used in arguments about programming, languages, and computation. ALGOL appears to be a durable model, and even flourishes under surgery—be it explorative, plastic, or amputative), and (e) it is tantalizingly inappropriate for many tasks we wish to program.

Of one thing I am sure, ALGOL does not owe its magic to its process of birth—by committee. Thus, we should not be disappointed when eggs, similarly fertilized, hatch duller models. These latter, while illuminating impressive improvements over ALGOL, bring on only a yawn from our collective imaginations. These may be improvements over ALGOL, but they are not successors as models.

Naturally we should and do put to good use the improvements they offer to rectify the weaknesses of ALGOL. And we should also ponder why they fail to stimulate our creative energies. Why, we should ask, will computer science research, even computer practice, worm, but not leap, forward under their influence? I do not pretend to know the whole answer, but I am sure that an important part of their dullness comes from focusing attention on the wrong weaknesses of ALGOL.

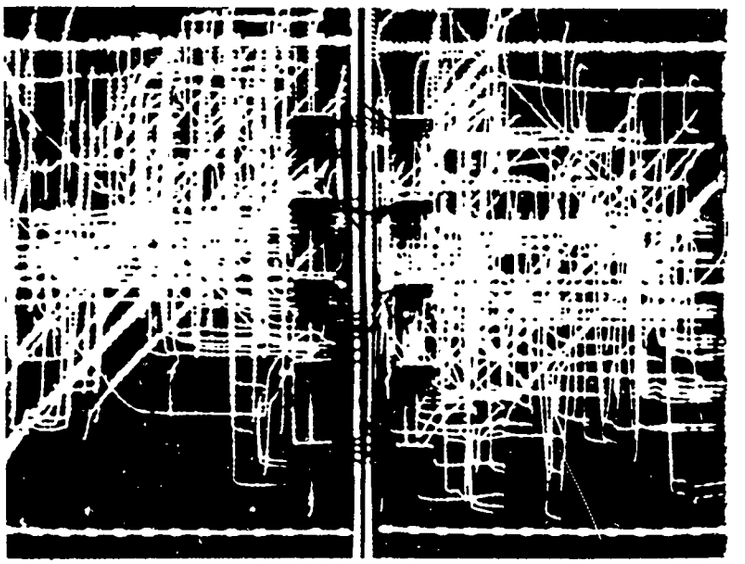
[1]This essay was delivered at the 21st National Conference of the Association for Computing Machinery as the first Turing Lecture. It is also reprinted in the Proceedings of that Conference.

We know that we design a language to simplify the expression of an unbounded number of algorithms created by an important class of problems. The design should be performed only when the algorithms for this class imposes, or are likely to impose after some cultivation, considerable traffic on computers as well as considerable composition time by programmers using existing languages. The language, then, must reduce the cost of a set of transactions to pay its cost of design, maintenance, and improvement.

Successor languages come into being from a variety of causes: (a) the correction of an error or omission or superfluity in a given language exposes a natural redesign which yields a superior language; (b) the correction of an error or omission or superfluity in a given language requires a redesign to produce a useful language; and (c) from any two existing languages a third can usually be created which (i) contains the facilities of both in an integrated form, and (ii) requires a grammar and evaluation rules less complicated than the collective grammar and evaluation rules of both.

With the above in mind, where might one commence in synthesizing a successor model which will not only improve the commerce with machines but will focus our attention on important problems within computation itself? I believe the natural starting point must be the organization and classifying of data. It is, to say the least, difficult to create an algorithm without knowing the nature of its data. When we attempt to represent an algorithm in a programming language, we must know the representation of the algorithm's data in that language before we can hope to do a useful computation.

Since our successor is to be a general programming language, it should possess general data structures. Depending on how you look at it this is neither as hard nor as easy as you might think. How should this possession be arranged? Let us see what has been done in the languages we already have. There the approach has been to: (a) define into the language a few "primitive" data structures, e.g., integers, reals, arrays (homogeneous in type), lists, strings, and files; and (b) on these structures provide a "sufficient" set of operations, e.g., arithmetic, logical, extractive, assignment, and combinational. Any other data structure is considered to be non-primitive and must be represented in terms of primitive ones. The inherent organization in the non-primitive structures is explicitly provided for by operations over the primitive data, e.g., the relationship between the real and imaginary parts of a complex number by real arithmetic. The "sufficient" set of operations for these non-primitive data structures are organized as procedures.



This process of extension cannot be faulted. Every programming language must permit its facile use for ultimately it is always required. However, if this process of extension is too extensively used algorithms often fail to exhibit a clarity of structure which they really possess. Even worse, they tend to execute slower than necessary. The former weakness arises because the language was defined the wrong way for the algorithm, while the latter is because the language forces over-organization in the data and requires administration during execution that could have been done once prior to execution of the algorithm. In both cases, variables have been bound at the wrong time by the syntax and the evaluation rules.

I think that all of us are aware that our languages have not had enough data types. Certainly, in our successor model we should not attempt to remedy this shortcoming by adding a few more, e.g., a limited number of new types and a general catch-all structure. Our experience with the definition of functions should have told us what to do: not to concentrate on a complete set of defined functions at the level of general use, but to provide within the language the structures and control from which the efficient definition and use of functions within programs would follow. Consequently, we should focus our attention in our successor model on providing the means for defining data structures. But this is not of itself enough. The "sufficient" set of accompanying operations, the contexts in which they occur, and their evaluation rules must also then be given within the program for which the data structures are specified.

We might list some of the capabilities that must be provided for data structures:

a
structure definition.
b
assignment of a structure to an identifier, i.e., giving the identifier information cells.
c
rules for naming the parts, given the structure.
d
assignment of values to the cells attached to an identifier.
e
rules for referencing the identifier's attached cells.
f
rules of combination, copy, and erasure both of structure and cell contents.

These capabilities are certainly now provided in limited form in most languages, but usually in too fixed a way within their syntax and evaluation rules.

We know that the designers of a language cannot fix how much information will reside in structure and how much in the data carried within a structure. Each program must be permitted its natural choice to achieve a desired balance between time and storage. We know there is no single way to represent arrays or list structures or strings or files or combinations of them. The choice depends on (a) the frequency of access, (b) the frequency of structure changes in which given data is embedded, e.g., appending to a file new record structures or bordering arrays, (c) the cost of unnecessary bulk in computer storage requirements, (d) the cost of unnecessary time in accessing data, and (e) the importance of an algorithmic representation capable of orderly growth so that clarity of structure always exists. These choices, goodness knows, are difficult for a programmer to make. They are certainly impossible to make at the design level.

Data structures cannot be created out of thin air. Indeed the method we customarily employ is the use of a background machine with fixed, primitive data structures. These structures are those identified with real computers, though the background machine might be more abstract as far as the defining of data structures are concerned. Once the background machine is chosen, additional structures as required by our definitions, must be represented as data, i.e., as a name or pointer to a structure. Not all pointers reference the same kind of structure. Since segments of a program are themselves structures, pointers such as "procedure identifier contents of (x)" establish a class of variables whose values are procedure names.

Truly, the flexibility of a language is measured by that which programmers may be permitted to vary, either in composition or in execution. The systematic development of variability in language is a central problem in programming, and, hence, in the design of our successor. Always our experience presents us with special cases from which we establish the definition of new variables. Each new experience focuses our attention on the need for more generality. Time sharing is one of our new experiences that is likely to become a habit. Time sharing focuses our attention on the management of our systems and the management by programmers of their texts before, during, and after execution. Interaction with program will become increasingly flexible, and our successor must not make this difficult to achieve. The vision we have of conversational programming takes in much more than rapid turn around time and convenient debugging aids. our most interesting programs are never wrong and never final. As programmers we must isolate that which

is new with conversational programming before we can hope to provide an appropriate language model for it. I contend that what is new is the requirement to make variable in our languages what we previously had taken as fixed. I do not refer to new data classes now, but to variables whose values are programs or parts of programs, syntax or parts of syntax, and regimes of control.

Most of our attention is now paid to the development of systems for managing files which improves the administration of the over-all system. Relatively little is focused on improving the management of a computation. Whereas the former can be done outside the language in which we write our programs, for the latter we must improve our control over variability within the programming language we use to solve our problems.

In the processing of a program text an occurrence of a segment of texts may appear in the text once but be executed more than once. This raises the need to identify both constancy and variability. We generally take that which has the form of being variable and make it constant by a process of initialization, and we often permit this process itself to be subject to replication. This process of initialization is a fundamental one and our successor must have a methodical way of treating it.

Let us consider some instances of initialization and variability in ALGOL. (a) Entry to a block. On entry to a block declarations make initializations, but only about some properties of identifiers. Thus, *integer x* initialize the property of being an integer but it is not possible to initialize the value of x as something that will not change during the scope of the block. The declaration *procedure* P (.); ; emphatically initializes the identifier P but it is not possible to change it in the block. Array A[1:n, 1:n] is assigned an initial structure. It is not possible to initialize the values of its cells, or to vary the structure attached to the identifier A. (b) *for* statement. The expressions, which I will call the step and until elements cannot be initialized. (c) the procedure declaration is an initialization of the procedure identifier. On a procedure call, its formal parameters are initialized as procedure identifiers are, and they may even be initialized as to value. However different calls establish different initializations of the formal parameter identifiers but not different initialization patterns of the values.

The choice permitted in ALGOL in the binding of form and value to identifiers has been considered to be adequate. However if we look at the operations of assignment of form, evaluation of form, and initialization as important functions to be rationally specified in a language, we might find ALGOL to be limited and even capricious in its available choices. We should expect the successor to be far less arbitrary and limited.

Let me give a trivial example. In the *for* statement the use of a construct like *value* E, where E is an expression, as a step element would signal the initialization of the expression E. *value* is a kind of operator that controls the binding of value to a form. There is a natural scope attached to each application of the operator.

I have mentioned that procedure identifiers are initialized through declaration. Then the attachment of procedure to identifier can be changed by assignment. I have already mentioned how this can be done by means of pointers. There are of course other ways. The simplest is not to change the identifier at all, but rather a selection index that picks a procedure out of a set. The initialization now defines an array of forms, e.g., *procedure array* P[1:k](f$_1$,f$_2$, ... f$_j$); . . . . begin . . . . *end;* . . . . ; *begin* . . . . *end;* The call P[i](a$_1$, a$_2$, . . a$_j$) would select the ith procedure body for execution. Or one could define a *procedure switch* P:=A,B,C and procedure designational expressions so that the above call would select the ith procedure designational expression for execution. The above approaches are too static for some applications and they lack an important property of assignment: the ability to determine when an assigned form is no longer accessible so that its storage may be otherwise used. A possible application for such procedures, i.e., ones that are dynamically assigned, is as generators. Suppose we have a procedure for computing

(a) $\sum_{k=0}^{N} C_k(N)X^k$ as an approximation to some function

(b) $f(x)=\sum_{k=0}^{\infty} C_k X^k$,

when the integer N is specified. Now once having found the $C_k(N)$ we are merely interested in evaluating (a) for different values of x. We might then wish to define a procedure which prepares (a) from (b). This procedure, on its initial execution, assigns, either to itself, or to some other identifier, the procedure which computes (a). Subsequent calls on that identifier, will only yield this created computation. Such dynamic assignment raises a number of attractive possibilities:

a
some of the storage for the program can be released as a consequence of the second assignment.
b
data storage can be assigned as the *own* of the procedure identifier whose declaration or definition is created.
c
The initial call can modify the resultant definition, e.g., call by name or call by value of a formal parameter in the initial call will affect the mind of definition obtained.

It is easy to see that the point I am getting at is the necessity of attaching a uniform approach to initialization and the variation of form and value attached to identifiers. This is a requirement of the computation process. As such our successor language must possess a general way of commanding the actions of initialization and variation for its classes of identifiers.

One of the actions we wish to perform in conversational programming is the systematic, or controlled, modification of values of data and text, as distinguished from the unsystematic modification which occurs in debugging. The performance of such actions clearly implies that certain pieces of a text are understood to be variable. Again we accomplish this by declaration, by initialization, and by assignment. Thus we may write, in a block heading, the declarations


```
real x,s;
arithmetic expression t,u;
```

In the accompanying text the occurrence of s := x + t; causes the value of the arithmetic expression assigned to t, e.g., by input, to be added to that of x and the result assigned as the value of s. We observe that t may have been entered and stored as a form. The operation + can then only be accomplished after a suitable transfer function shall have been applied. The fact that a partial translation of the expression is all that can be done at the classical "translate time" should not deter us. It is time that we began to face the problems of partial translation in a systematic way. The natural pieces of text which can be variable are those identified by the syntactic units of the language.

It is somewhat more difficult to arrange for unpremeditated variation of programs. Here the major problems are the identification of the text to be varied in the original text and how to find its correspondent under the translation process in the text actually being evaluated. It is easy to say: execute the original text interpretively. But it is through intermediate solutions lying between translation and interpretation that the satisfactory balance of costs is to be found. I should like to express a point of view in the next section which may shed some light on achieving this balance as each program requires it.

Even though list structures and recursive control will not play a central role in our successor language, it will owe a great deal to LISP. This language induces humorous arguments among programmers, often being damned and praised for the same feature. I should only like to point out here that its description consciously reveals the proper components of language definition with more clarity than any language I know. The description of LISP includes not only its syntax, but the representation of its syntax as a data structure of the language, and the representation of the environment data structure also as a data structure of the language. Actually the description hedges somewhat on the latter description, but not in any fundamental way. From the above descriptions it becomes possible to give a description of the evaluation process as a LISP program using a few primitive functions. While this completeness of description is possible with other languages, it is not generally thought of as part of their defining description.

An examination of ALGOL shows that its data structures are not appropriate for representing ALGOL texts, at least not in a way appropriate for descriptions of the language's evaluation scheme. The same remark may be made about its inappropriateness for describing the environmental data structure of ALGOL programs. I regard it as critical that our successor language achieve the balance of possessing the data structures appropriate to representing syntax and environment so that the evaluation process can be clearly stated in the language. Why is it so important to give such a description? Is it merely to attach to the language the elegant property of "closure" so that bootstrapping can be organized? Hardly. It is the key to the systematic construction of programming systems capable of conversational computing.

A programming language has a syntax and a set of evaluation rules. They are connected through the representation of programs as data to which the evaluation rules apply. This data structure is the internal or evaluation directed syntax of the language. We compose programs in the external syntax which, for the purposes of human communication, we fix. The internal syntax is generally assumed to be so translator and machine dependent that it is almost never described in the literature. Usually there is a translation process which takes text from an external to an internal syntax representation. Actually the variation in the internal description is more fundamentally associated with the evaluation rules than the machine on which it is to be executed. The choice of evaluation rules depends in a critical way on the binding time of the variables of the language.

This points out an approach to the organization of evaluation useful in the case of texts which change. Since the internal data structure reflects the variability of the text being processed, let the translation process choose the appropriate internal representation of the syntax, and a general evaluator select specific evaluation rules on the basis of the syntax structure chosen. Thus we must give clues in the external syntax which indicate the variable. For example, the occurrence of *arithmetic expression* t; *real* u, v; and the statement u := v/3*t; indicates the possibility of a different internal syntax for v/3 and the value of t. It should be pointed out that t behaves very much like an ALGOL formal parameter. However the control over assignment is less regimented. I think this merely points out that formal-actual assignments are independent of the closed sub-routine concept and that they have been united in the procedure construct as a way of specifying the scope of an initialization.

In the case of unpremeditated change a knowledge of the internal syntax structure makes possible the least amount of retranslation and alteration of the evaluation rules when text is varied. Since one has to examine and construct the data structures and evaluation rules entirely in some language, it seems reasonable that it be in the source language itself. One may define as the target of translation an internal syntax whose character strings are a sub-set of those permitted in the source language. Such a syntax, if chosen to be close to machine code, can then be evaluated by rules which are very much like those of a machine.

While I have spoken glibly about variability attached to the identifiers of the language, I have said nothing about the variability of control. We do not really have a way of describing control, so we cannot declare its regimes. We should expect our successor to have the kinds of control that ALGOL has—and more. Parallel operation is one kind of control about which a good deal of study is being done. Another one, just beginning to appear in languages, is the distributed control which I will call monitoring. Process A continuously monitors process B so that when B attains a certain state A intervenes to control the future activity of the process. The control within A could be written *when* P *then* S; P is a predicate which is always, within some defining scope, under test. Whenever P is *true*, the computation under surveillance is interrupted and S is executed. We wish to mechanize this construct by testing P whenever an action has been performed which could possibly make P *true*, but not otherwise. We must then, in defining the language, the environment, and the evaluation rules, include the states which can be monitored during execution. From these primitive states others can be constructed by programming. Knowing these primitive states, arrangement for splicing in

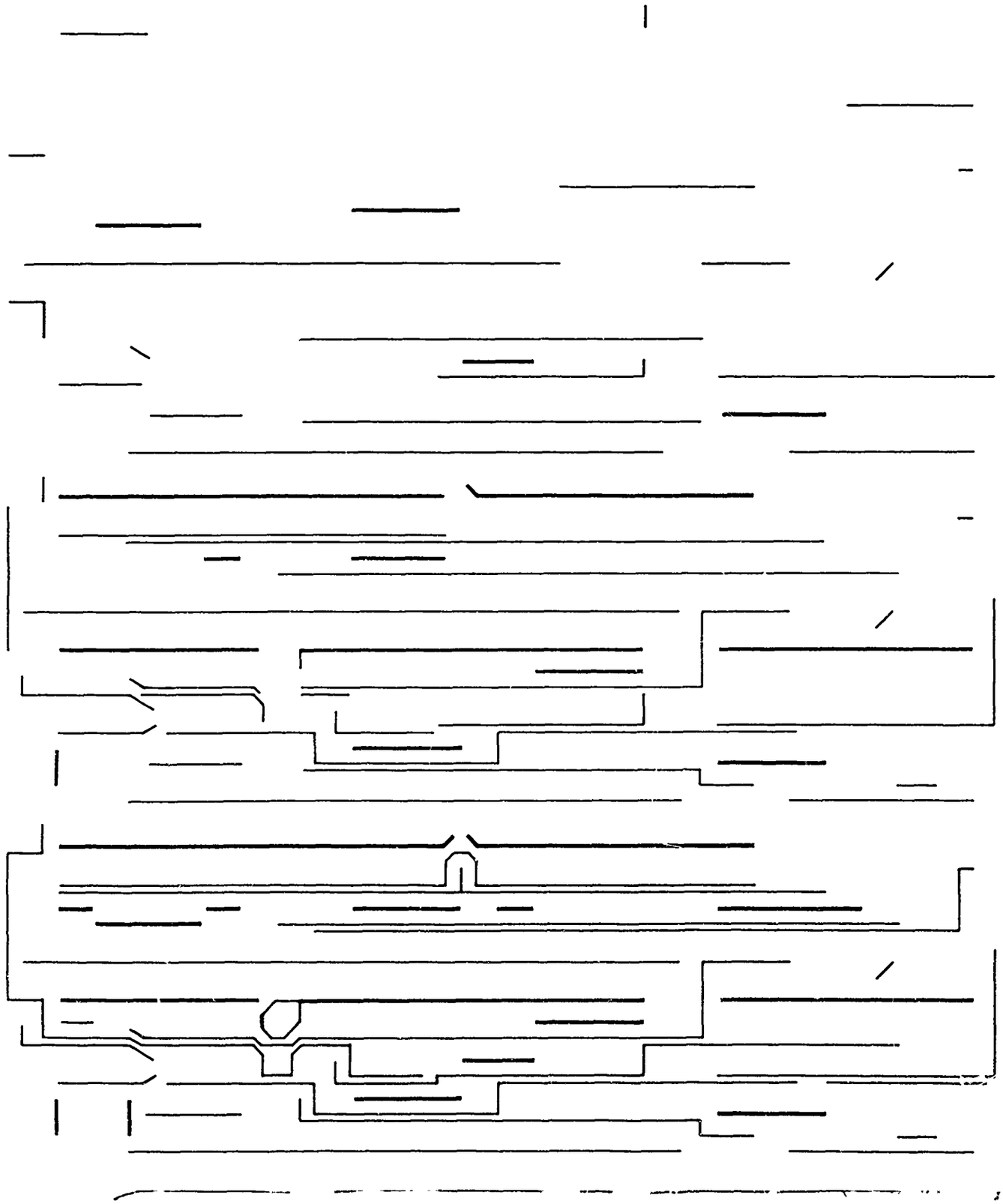
testing at possible points can be done even before the specific predicates are defined within a program. We may then trouble-shoot our programs without disturbing the programs themselves.

Within the confines of a single language an astonishing amount of variability is attainable. Still all experience tells us that our changing needs will place increasing pressure on the language itself to change. The precise nature of these changes cannot be anticipated by designers since they are the consequence of programs yet to be written for problems not yet solved. Ironically it is the most useful and successful languages that are most subject to this pressure for change. Fortunately, the early kind of variation to be expected is somewhat predictable. Thus, in scientific computing the representation and arithmetic of numbers will vary, but the nature of expressions will not change except through its operands and operators. The variation in syntax from these sources is quite easily taken care of. In effect the syntax and evaluation rules of arithmetic expression is left undefined in the language. Instead syntax and evaluation rules are provided in the language for programming the definition of arithmetic expression, and to set the scope of such definitions.

The only real difficulty in this one-night stand language definition game is the specification of the evaluation rules. They must be given with care. For example, in introducing this way the arithmetic of matrices, the evaluation of matrix expressions should be careful of the use of temporary storage and not perform unnecessary iterations. A natural technique to employ in the use of definitions is to start with a language X, consider the definitions as enlarging the syntax to that of a language X', and give the evaluation rules as a reduction process which reduces any text in X' to an equivalent one in X. It should be remarked that the variation of the syntax requires a representation of the syntax, preferably as a data structure of X itself.

Programming languages are built around the variable, its operations, control, and data structures. Since these are concepts common to all programming, general language must focus on their orderly development. While we owe great debt to Turing for his simple model, which also focused on the important concepts, we do not hesitate to operate with more sophisticated machines and data than he found necessary. Programmers should never be satisfied with languages which permit them to program everything, but to program nothing of interest easily. Our progress, then, is measured by the balance we achieve between efficiency and generality. As the nature of our involvement with computation changes—and it does—the appropriate description of language changes, our emphasis shifts. I feel that our successor model will show such a change. Computer Science is a restless infant and its progress depends as much on shifts in point of view as the orderly development of our current concepts. None of the ideas presented here are new, they are just forgotten from time to time.

I wish to thank the Association for the privilege of delivering this first Turing Lecture. And what better way is there to end this lecture than to say that if Turing were here today he would say things differently.



In the beginning was the word. I am the Alpha and Omega. The growth of Lear. He grew old and mad. There's growth for you.

Thomas Wolfe
Look Homeward, Angel

Herbert A. Simon, R. K. Mellon Professor of Computer Sciences and Psychology

Reflections on Time Sharing From a User's Point of View

Ten years ago, programs were commonly debugged (and sometimes also run) by a user who sat at the computer console and interacted with it "on line"—queried the contents of particular memory addresses and received an immediate answer, or stepped through the program instruction by instruction. As computers became bigger and faster, manual operation from the console became more and more wasteful of the central processor's time, for the processor was almost always idle while waiting for the slow response of the human operator.

The subsequent era of batch processing was distressful to programmers, because it did not eliminate the man-computer imbalance, but simply reversed its direction. Now the programmer had to wait for the response of the computer, and unless he could switch to another task while waiting, much of his time was idle. Thence arose the idea of keeping the programmer busy by allowing a number of users to share the time of the central processor, and perhaps to share high-speed memory as well.

Since present-day computers are typically built around a large, single processor, users do not literally "share" time, do not have simultaneous access to the processor. Instead, the processor serves the demands of the several users in turn, sharing its processing capacity among them. With an appropriate balance of number of users to processing power, both users and computer may be kept reasonably occupied. Central memory may actually be shared, if several users' jobs are held in memory simultaneously. Alternatively, users may share secondary memory. That is, the processor may "swap" from secondary into central memory the program and data of the user who is to receive the next slice of processing time. Often a time-sharing system provides a combination of core-sharing and swapping to handle the collective memory requirements of its users.

Since time-sharing has generally been proposed as a means for on-line conversation between computer and user, the design of time-sharing systems has usually aimed at the shortest possible turnaround time compatible with reasonable economies of memory swapping. Design based on these considerations has been useful for exploring the potentialities of time sharing, and the organizational problems that need to be solved in constructing time-sharing software.

As time-sharing systems become a larger part of the workaday world of the computer user, however, a broader and deeper design philosophy will be needed. Such a design philosophy can be obtained by paying closer attention to the diversity of purposes to which

time-sharing and on-line techniques may be directed, and to the characteristics of the human component in the time-sharing system (which is fundamentally and essentially a man-machine system whose performance depends on how effectively it employs the human nerveware as well as the computer hardware).

How well do existing general-purpose time-sharing scheduling algorithms take care of: (a) end-use considerations or (b) the information processing characteristics of their human users?

1

In these systems the unit of interaction is almost always taken to be a sequence that begins with a message from the human component and ends with a response from the computer. Since, in a "conversation" between man and computer, there is symmetry between the two components, why not, instead, take as the unit of interaction a sequence that begins with a message from the computer and ends with the response from the human? Quite different software design (as I shall try to indicate below) might result from this changed point of view.

Is either point of view, in fact, correct? Isn't the system really a closed feedback loop in which the human user proposes tasks to the computer and the computer to the human?

2

In current practice, the basic "time slice" is defined, again asymmetrically, in terms of the relation between the cost and frequency of swapping between hardware memories. Would a different system design emerge if explicit attention were also paid to the costs of swapping in human memory? Isn't a major objective of time-sharing to avoid the necessity of the human switching in and out of context?

3

Most existing designs for time-sharing software are heavily biased toward use of the system in a conversational mode—i.e., at rates of interaction between human user and computer comparable to normal conversational rates. How would the design change if many or most users needed interaction on, say, a ten minute or fifteen minute cycle? (This is the kind of interaction that the Carnegie G-21 system, for example, has provided.)

4

Is it correct to assume that the faster the turnaround, the better for the human user? Not only does this assumption ignore the last two considerations mentioned—human swapping costs, and the bias toward a conversational mode—but it is ambiguous with respect to the concept of turnaround. Does "fastest turnaround" mean most rapid response to each individual inquiry (e.g., a one-line instruction), or does it mean most rapid completion, by the computer, of the substantive job of interest to the human user (e.g., inverting a matrix)?

To answer questions like these, we must have a clearer idea of what the user wants.

To say that a user is "on line" says very little about the requirements that must be met by the system if he is to use it effectively. Lacking a knowledge of the different kinds of uses, their frequency and importance, the designer of software must build a general-purpose system. A price is paid for the flexibility that makes a system suitable for a variety of purposes—it is always less efficient for any single one of those purposes than if it had been designed specifically for it. Of course when flexibility is sacrificed for single-purpose efficiency it almost always turns out in practice that the purpose for which the system is designed is not the exact purpose for which it is needed. When we are designing a "public utility," we must be willing to pay the price of flexibility.

To design a system as a flexible, multi-purpose facility, however, requires explicit attention to the range of uses to which it is to be put. It will not do for the designer simply to put in the forefront of his attention one or two of the possible uses, and to assume that if the system serves those it will serve the others.

Even a cursory survey of existing time-sharing systems shows that there are at least seven distinguishable classes of on-line uses, and that different systems have been designed with a variety of these uses in mind:

1
Real-time acceptance of data and maintenance of displays, the former in such applications as missile tracking and reduction of data from laboratory experiments, the latter in design and teaching machine applications (it is usually inefficient to mix microsecond real-time demands of these kinds with a general purpose system, if the real-time demands require responses in times close to the basic swap time. Swapping costs will be exhorbitant unless there is a powerful interface to provide buffering and relax the time constraints on the central system);
2
Real-time operation at human conversational rates for such applications as text-editing, idiot-error debugging, and answering simple inquiries (e.g., reservations inquiries);
3
Desk calculator use, offering modest computational power—*not* including solution of large linear programming problems or partial differential equations at conversational rates;
4
Debugging beyond detecting obvious syntax errors;
5
Storage and retrieval of large amounts of information—most business data processing applications;
6
"Production", using substantial amounts of central processor time to provide substantive information to the user;
7
Evaluating "cases" with a model, by studying the model's behavior under changes of parameters and subroutines.

The classification is not intended to be exhaustive, but illustrative of the characteristics of various kinds of demands. In some cases, the categories overlap. In general, though not invariably, the early items on the list demand less central processing than the later items; while the tolerable delay in response is greater in the later than in the earlier.

In my subsequent comments, I will have nothing further to say about data acquisition and display maintenance, but will assume that the system is so buffered that the tasks calling for very high-frequency response will look no different to the main system than conversational-rate tasks.

Categories 2 and 3 include tasks where one would like conversational-rate response to messages from the human user. For these categories the currently-used design approaches seem to work reasonably well. The swap-time characteristics of the hardware determine the minimum "slice"—call it S—that can be used without incurring major swapping costs. (I shall have something to say later about interactions between swapping costs and the nature of the computing load.) The characteristics of the human user—the normal rate of human speech—determine the desired terminal turnaround or cycle time—call it T. Then, the system will accommodate at most $U=T\ S$ simultaneous users in the conversational mode, with turnaround not exceeding T and slice not less than S. The nature of the scheduling algorithm may alter U slightly, but cannot change it fundamentally, for on the average $T\ S$ users can be given a slice of length S every T seconds.

This estimate of the capacity, U, assumes that the central processor will typically be able to complete a response to a user in S seconds or less of actual processing time, an assumption that is seldom valid. Usually, the shortest slice that is tolerable from the standpoint of swapping efficiency, S, is much less than the average slice R that is needed to service a request. *Under these (typical) conditions, R, not S, should govern the system design, since response at conversational speeds can only be maintained for $U'=T\ R$ users, irrespective of the scheduling algorithm.* In particular, no improvement can be obtained in turnaround, and there will be an actual loss in central processor efficiency (due to increased swapping) if the slice is reduced below R. The reduction will raise turnaround time, for it will mean that some users will have to wait for more than one cycle for a response (i.e., at capacity, for more than the designed cycle time, T).

For these reasons, the number of on-line users the system can accommodate simultaneously at conversational rates is more or less independent of the

minimum tolerable slice, S (assuming this to be less than *R*), *but varies inversely with the average computing demand per interaction, R.* No scheduling magic can relax this iron law.

While in categories 2 and 3 we might conceivably stipulate some fixed *R*, and limit any single demand to that amount of computing, uses in the remaining categories can demand any amount of computing—up to hours. Of course, the system can set an arbitrary upper limit; our phrase "can demand" means that the user needs that amount of computing to get an answer to the question he is asking, and not simply to a question that has been artificially fragmented to fit the scheduling algorithm.

In general, existing time-sharing scheduling systems solve the problem of lengthy requests by abdicating responsibility for them. Sometimes, they define a time slice, and guarantee to the user who limits his requests to a small multiple of that time slice a response at conversational rates. Often, they do not even provide this guarantee, but allow the shortest response time gradually to increase as the total demand on the system increases. The user demanding more central processing is regulated by impedimenta—the larger his demand, the worse the service he can expect.

Scheduling algorithms embodying these characteristics have a certain aura of "fairness" about them; but it may be questioned whether they are adapted to the needs of users. Do they retain their plausibility when we take a systems approach, instead of a local approach, to the time allocation problem? To get a broader view, let us step back and examine a little closer the nature of a conversation between a man and a computer—or, for that matter, between two men.

If one person is much better informed on the topic of the conversation than the other, the balance in computing capacities can still be maintained if the former adopts the role of informing the latter (by "lecturing" or by answering questions). The inter-action can still go on at a conversational rate, although unless the controlling participant (lecturer or questioner) receives and reacts to feedback from the other, he is likely to swamp the capacity of the other's processor. The receiver may cease listening, or the person being questioned may no longer be able to give the answers in real (conversational) time. ('Quick! What are the prime factors of 3,628,800?'). Then the conversation is punctuated by silences, which serve as signals either to change its character or to terminate it.

In particular, if the questioner does not receive answers promptly, his own processor is idle much of the time, and he will likely say, "Why don't you run down the answer on that, and we'll talk again tomorrow," or, "Send me a memo on it." On the other hand, the answers to questions may give the questioner so much food for thought that the long silences will be his. In that case, the other person may say: "If you have any questions, I'll be here tomorrow."

Not all reactions to unbalanced conversations are adaptive. If the silences are longer than normal, but the conversation rate remains about half, say, of the usual one, the person whose processing capacity is under-utilized may simply remain in the situation, slightly bored.

A more subtle disfunction occurs when one of the participants processes information inefficiently because it is not presented to him at an appropriate rate or in an appropriate format. To take a very simple example, suppose that A is adding numbers that are being read to him by B. If B reads more slowly than A can add, A will likely stay in the situation, but use his time inefficiently. It would be far better if A could turn off and do another task while the numbers are recorded, then add them all at once.

These possibilities of imbalance in man-man or man-machine conversation are not imaginary. There are

---

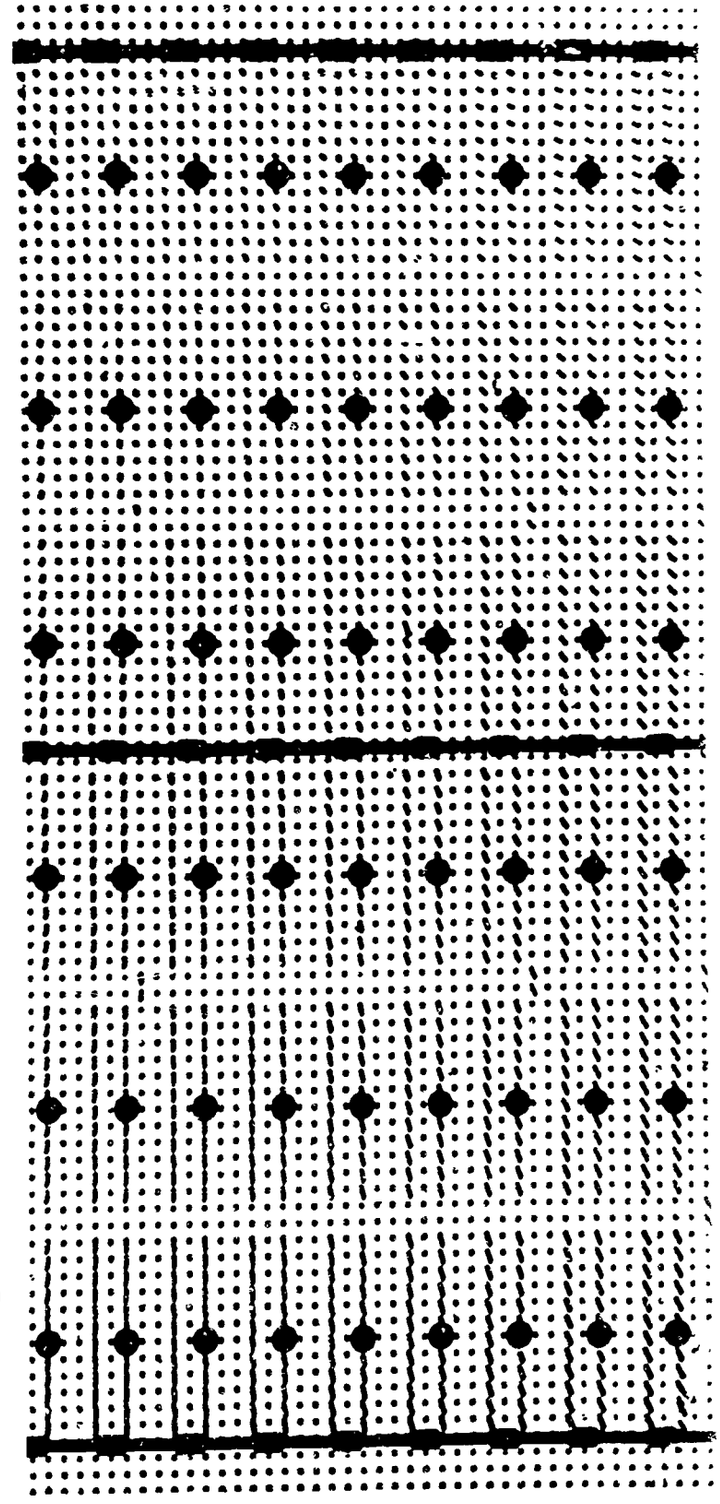
In a casual conversation two humans alternate in speaking a few sentences at a time, the joint rate usually being less than one sentence every four seconds. When the two persons are approximately matched intellectually and are taking symmetric roles in the conversation, the system is balanced. Each person has as much time as the other to process the communications directed to him, and to construct his replies.

few programmers who have not had experiences of being "slaved" to teletypes or consoles—where information was not arriving fast enough to occupy them fully, but where the intervals were not long enough to allow them to swap between the conversational task and some other one.

There is a good deal of *a priori* probability, and some empirical evidence, for the notion that changes in the form of interaction can have a major impact on problem-solving style. (P. Soelberg, private communication.) Two groups of subjects were given a task amounting essentially to discovering the maximum of a function of several variables in the presence of noise. The first group was permitted to evaluate the function repeatedly for different sets of values of the independent variables, selecting a new set of values after receiving the results of the previous choice. The second group was required to make a batch of, say, ten evaluations at a time, receive the results of that batch, and then permitted to specify a new batch. Subjects in the first group almost without exception conducted local hill-climbing explorations of the function, taking account on each trial of the results of the past couple of trials. Subjects in the second group selected sets of values of the independent variables in such a way as to carry out a systematic exploration of the shape of the function.

Either of these two procedures might be the more efficient for particular classes of functions. The important point is that they are different, and radically different, and that the choice was made between them not on a deliberate, rational basis, but by the impact of the stream of feedback information. There is no reason to suppose that this situation was idiosyncratic, hence there is every reason to suppose that the nature of the system's response to a human participant can have decisive effects on his problem-solving methods. How can scheduling algorithm be designed with concern for such effects?

In the present state of our psychological knowledge, it would be unreasonable to expect the designer of a scheduling algorithm to estimate optimum modes of interaction for all future uses of the system. Among



other difficulties, he has no way of predicting exactly what those uses are. It is perhaps less unreasonable for the designer to construct an algorithm that will allow the user to try to devise for himself a reasonably efficient mode of interaction with the computer for a wide variety of possible uses.

One possible rule of thumb for the design is to require *neither the human nor the machine component of the system to respond at rates either above or very much below their processing capacity.* This condition is easier satisfied for computer than user, provided the basic time slice is not too short, since the former can always use its "idle" time productively for computing "background" jobs. The linkage of human user with machine should be "conversational" in the strict sense of the term only when the processing demands are at the conversational level for both. Otherwise, an alternative mode of interaction should be employed.

A second possible design principle is that turnaround times *are ultimately important only for "finished" jobs.* If a user wants the square root of two calculated to twenty places, he will usually be just as pleased to receive the whole answer after four seconds as to receive one digit each two-tenths of a second. Since a user generally wants an answer, however, in order to ask another question (if nothing else, to go on with his next piece of research), defining "finished" jobs is a matter of some subtlety.

A third guide to the design is that *the human, like the computer, has minimum swap times, which we can estimate roughly from everyday experience* even if we do not know their precise values. We have already mentioned one time parameter of the human system—in the conversational mode, he should receive responses within a couple of seconds. A second critical parameter is a little vaguer: how much time does it take to change context to another task? As a rough guess, free time in slices of less than ten minutes can generally not be used efficiently except on tasks closely connected with the one to which attention is paid. An interval of ten minutes or more, particularly if the time when the interruption will occur is predictable, can usually be used fairly efficiently. (This probably means that humans can *learn* to use such an interval efficiently with a little experience.) For a major change of context to a new complex task, the swapping costs may be excessive unless several hours are free. This will, of course, be particularly true if the task has to be performed in a different place.

If these conjectures are at all correct, then a user might be given the alternatives of (1) operating in a conversational mode, (2) operating with a ten to twenty minute turnaround time, or (3) submitting tasks that will be processed in, say, a day. (For any system operating within its capacity, the maximum turnaround time need not be much longer than a small multiple of the time required to process the biggest job.) In any of these modes, he should be able to form firm expectations as to when (at latest, and perhaps, also at soonest) a task will be finished.

It will not only be useless, but absolutely harmful, (1) to provide service that is slower than conversational, but gives faster than ten minute's turnaround, and (2) service that is slower than ten to twenty minutes, but of the order of magnitude of an hour. (If the precise numbers given here are wrong, the general principle, in terms of our swapping analysis, is still right. With a little trouble, it should be possible to make reasonable estimates of the parameters.) *It will be harmful, also, to provide unpredictable service in any mode.*

A scheduling system efficient in terms of processing capacity, and meeting the specifications just outlined, could be relatively simple—simpler than most existing time-sharing algorithms. It could be based on two time-slice parameters: R_c, the maximum processing time a user will receive in a single response in the conversational mode; and R_d, the maximum processing time he can demand in a single response in the ten-minute ("debugging") mode. There would usually need to be an upper limit on processing time for all other tasks (background), with provision for exceptions with administrative approval.

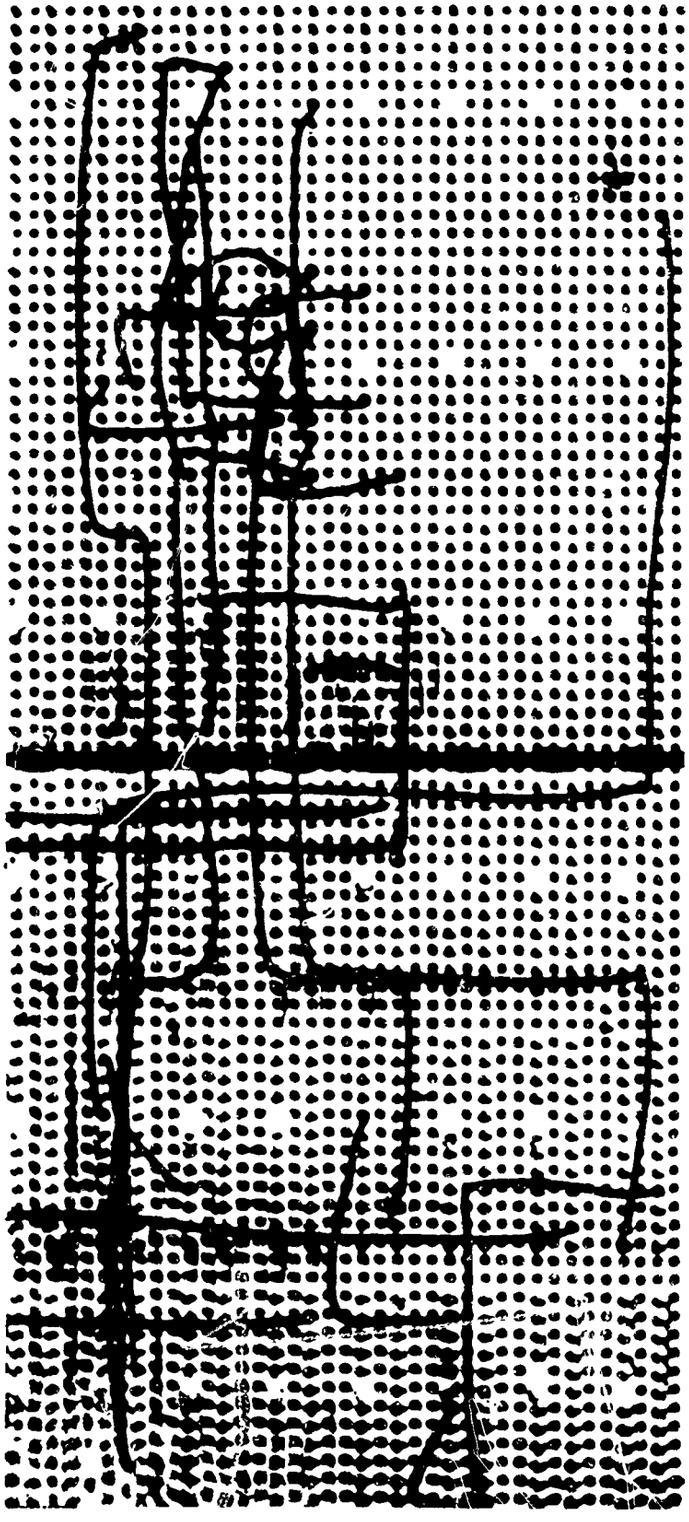
The classification of a request can be made to depend on information from the user, about the particular subsystems the request calls upon, expected processing

time, or some combination of these. The three classes of requests are processed prior to others, then all waiting debugging requests, then background. When a request has reached the head of its queue, and no higher-priority request is waiting, it is processed to completion or to the time limit for its class, whichever occurs first.

No elaborate statistics need be kept on how much processing each incomplete task has received. At any given moment all tasks except the one (or few, in a multi-processor) that is actually being processed have either not received any time at all, or they have received all they are going to get. The priority system consists simply of the three lists of unprocessed tasks, in order of receipt. As experienced production schedulers will recognize, the scheme minimizes in-process inventory, hence giving near-minimum turnaround times provided that each priority class is reasonably homogeneous (as it will be under this scheme).

The scheme will only work if the time-slice parameters, R_C and R_d are set so that the system is not loaded beyond its capacity. There may need to be some provision for *occasional* adjustment of these parameters, but rapid feedback adjustment might introduce intolerable uncertainties unless an appropriate way is found to inform users before they submit tasks. This limitation is not peculiar to the proposed scheme—any scheduling algorithm that promises performances exceeding to aggregate the production capacity of the system will simply fail to keep its promises. The usual way in which time-sharing algorithms fail is by reneging on promised turnaround times, or by shutting low priority users (if there are priority classes) out of the system entirely.

Consider now the setting of the parameters. There is needed an estimate, first, of what part of the system capacity is to be devoted to conversational, what part to debugging, and what part to background demands. These percentages may need to vary automatically from one part of the day to another, and will need to be adjusted on the basis of experience. The basic constraint on the system is of the form $U_i=T_i/R_i$, which can now be applied to the three modes separately. The



parameter T_i will be twenty seconds for the conversational mode; and twenty minutes for the debugging mode. (These numbers are just proposed for illustration; they include a complete cycle from one initiation by the user to the next, hence include the user's response time as well as the processor's.)

Next, the choice can be made between servicing a large number of users simultaneously, with small R_i, of a smaller number, with larger R_i (since UxR is a constant). At this point existing scheduling algorithms generally abdicate in refusing to limit the number of simultaneous users. They usually ration time, as already suggested, by the degradation of T. The deterioration almost never affects different users equally, and sometimes does not even affect them predictably; thereby, guaranteeing again that Justice is indeed blind, and stochastic as well.

For most purposes, the number of users to be served over a day's time must be considered as given for the scheduler. (The question of allocating the overall capacity of the system among users cannot be considered in this paper. It can be stated unequivocally, however, that such allocation can never be accomplished in a satisfactory manner by any sort of scheduling algorithm—i.e., an algorithm determining when particular tasks are processed.) It may be necessary, because of capacity constraints, to limit use during certain parts of the day to certain of the three priority classes.

Given the number of simultaneous users, U_i, in a class, i, and the time slice, T_i, for that class, the parameter R_i is fixed, and determines, in turn, how elaborate are the responses that users may expect, at maximum, in that mode of use. This principle may lead to an important economy in designing the conversational mode, for it may be necessary to give the user access to no more elaborate facilities than he can use within his time constraints. Hence, interruptions for conversational mode may require less complete and elaborate swapping than interruptions for the debugging and background modes. Since the former swaps will be by far the most frequent, this may permit a large reduction in swapping costs. In many uses, for example, searching a large data store, this economy may not be available since each user may require a large amount of high-speed memory. A further elaboration of the design might achieve some swapping economies by dividing the conversational traffic into two streams, only one of which requires swapping. But our purpose here is to outline a design viewpoint, not to elaborate details of design.

Limiting the sorts of tasks that will be performed in conversational mode will go a long way toward inducing users to make a realistic choice of mode. Perhaps the simplest way to enforce the restrictions is to transfer automatically a program that exceeds the time slice for a mode to the next lower mode. Thus a user who indicated a desire for a response at conversational speed, but who posed a task requiring a time slice longer than R_c would have his incomplete job stored at the end of the queue for debugging tasks, and would receive a message notifying him of this.

Since the time slice will vary inversely with the number of simultaneous users the system is designed to handle, if many users are to be accommodated each will have the experience of conversing with a machine of modest computing power, so that he can only ask it relatively simple questions in conversational mode, and must send it away, in debugging or background mode to find the answers to more difficult questions. If fewer users are accommodated, the conversational power of the computer will be larger for each.

A simple numerical example will illustrate how the system would operate. Let us assume that 500 users are to be accommodated in conversational mode (T−20 seconds), 60 in debugging mode (T=20 minutes), and an indefinite number in background mode. Let us assume, further, that a time slice of 4 ms is reasonable for the sorts of editing, simple debugging, querying, and desk calculator tasks that are to be handled conversationally. Then the conversational mode will absorb (500.4)/20 ms/sec of the processing time, that is, 100 milliseconds per second, or ten percent of the total available. Similarly, if the debugging mode is allowed a time slice of 10 seconds, it will absorb (60.10)/20 sec/min, or 500 milliseconds per second would remain for background. If background

tasks averaged two minutes in length, one could be completed about every five minutes.

At the beginning of each second, for instance, the processor would interrupt and swap out the job on which it was working, then execute all the tasks on the conversational queue, then all those on the debugging queue then return to its interrupted task. If a background task were finished during that second, another would be started. Since, on the assumptions, an average of 25 conversational tasks and a negligible number of debugging and background tasks would arrive each second, the number of interruptions per second would average just slightly more than twenty-five, almost all of them in the conversational mode.

Scheduling systems have sometimes been designed on the basis of assumptions as to the frequency distributions of tasks by time required. Such reasoning is generally circular. Users will adjust rapidly to the upper limits of times allowed for various classes of service. Thus, if there is a one minute time limit for tasks of a certain class, it will almost always be observed that the system is currently working on a task that takes one-half to three-quarters of a minute. There are two reasons for this, one psychological, one statistical. The psychological reason is that users will design tasks to obtain as much time as possible, short of being bumped to a lower priority. The statistical reason is that the longer tasks in any class in fact absorb the bulk of the processing time available for that class. For these reasons, statistics of task length reflect the scheduling algorithm quite as much as they reflect the needs of users; hence, they are relatively useless for designing the system.

The reflections in this paper have centered around the notion that in designing a man-computer system, both human user and computer can be specified in terms of the same parameters—principally, processing rate and memory swap time—although man and computer will have quite different numerical values for each parameter. Viewing a man-computer system as a more or less symmetric structure in which the machine initiates tasks for the man just as the man initiates tasks for the machine suggests approaches to the control of time-sharing systems quite different from those typically taken in the past.

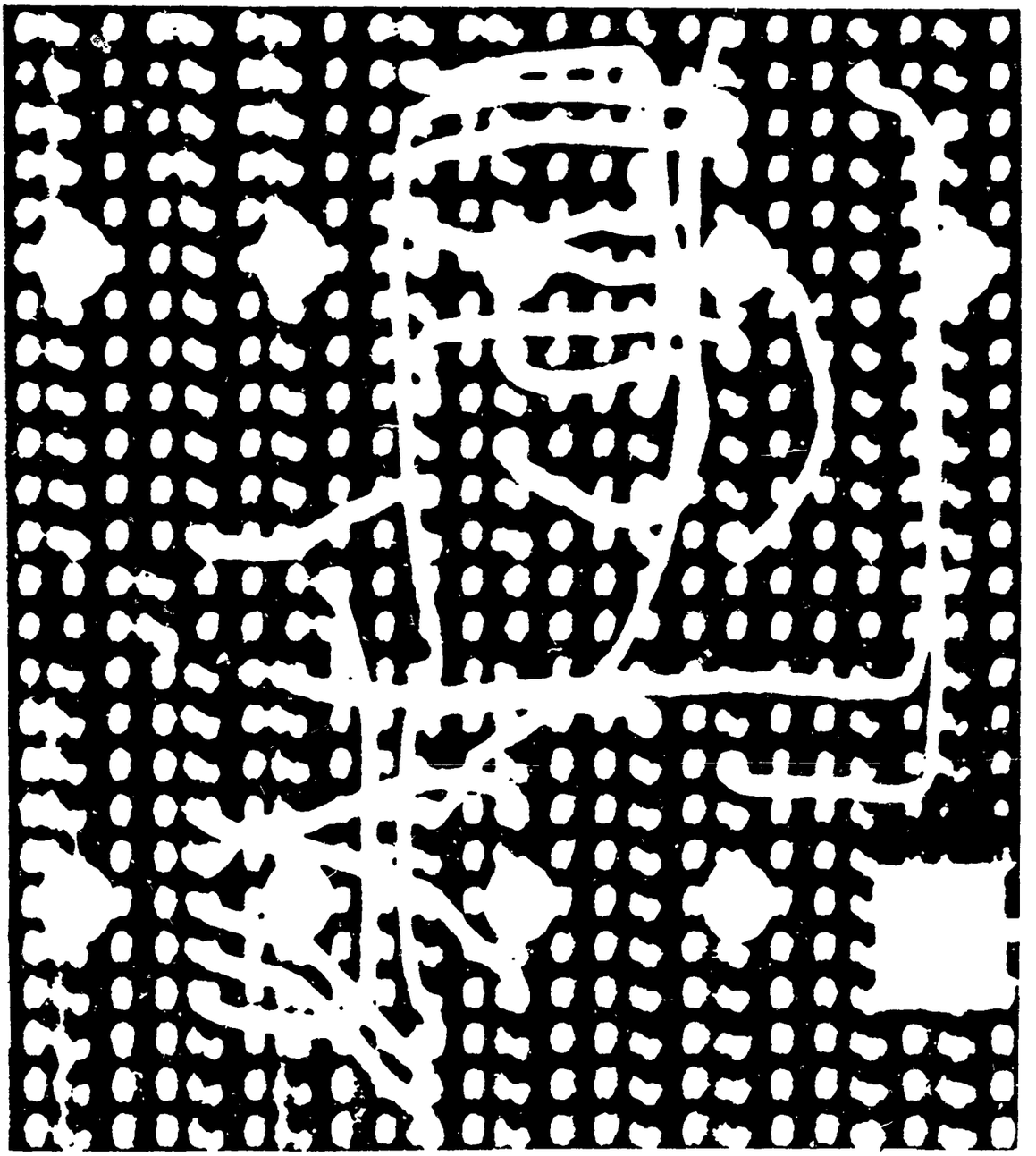
From the design standpoint, the critical derived parameters of such a system are turnaround times that will permit the human users to employ their time efficiently, and computer time slices that will allow tasks of appropriate size to be assigned to the processor. Regardless of the design philosophy, the ratio of turnaround time to processing time slice determines the number of users who can be accommodated simultaneously in particular mode by a given fraction of machine capacity. No scheduling system can accommodate more users without shortening the time slices or allowing degradation of turnaround times. If the system cannot handle the existing number of potential users, then degradation of service can only be avoided by allocating access to the system in some way. Scheduling algorithms cannot provide satisfactory allocation schemes, since the attempt to combine the allocation with scheduling will give the system undesirable operating characteristics.

These basic reflections on the principles of design have been illustrated by a sketch of a possible control system. The word "sketch" should be emphasized, for the analysis would have to carry to much greater detail to provide a workable system, and even to demonstrate that the system would really have the desirable properties it appears to offer. The purpose of the sketch is not to recommend this particular control system, but to outline how this philosophy of design could actually be applied.

If a control system can be designed along these lines, it promises to be simpler in operation, and require less memory and less calculation of priorities, than time-sharing algorithms that have been described in the literature. As we have seen, there are a number of directions in which it might be elaborated, but elaboration would not necessarily improve its operation. Its very simplicity should guard it against the kinds of unanticipated consequences that arise in systems that are excessively complicated and subtle. A half century of experience with production and process controls has taught the field of control engineering that simplicity is indeed a very important virtue for a control system.

Today is not yesterday.— We ourselves change.—
How then, can our works and thoughts, if they are
always to be the fittest, continue always the same.

Thomas Carlyle



Jesse T. Quatse, Manager, Engineering Development

Generality and Computer Design

Like any other machine, a computer system earns its upkeep by performing the specific collection of tasks for which it was designed. The efficiency and speed with which it performs these tasks is dependent upon how well it matches the task mix. Beneath the surface of the computer system, task fragments become tasks for sub-systems, and beneath that level, sub-systems are composed of sub-sub-systems. A good system match is therefore the gross effect of good component to task matching at many levels. It is not usually an easy effect to achieve.

In order to visualize the difficulty, one can imagine a "perfectly monotonous" task environment in which a particular computer system is to spend its operating life. In this environment, everything important is known about all tasks and no task ever changes in any way. All details of the system designed for this particular perfectly monotonous environment could be safely fixed once and for all at design time. All sub-systems could be tailored precisely to fit the task environment without the danger that the speed or cost of any sub-system is inappropriate to the task performed.

For example, time consuming memory accesses could be eliminated by permanently "wiring in" all programs and program constants. Data transmission sub-systems could be designed to accommodate a fixed number and type of tape units, disk units, etc. Normally, the task environment is reduced to a relatively small number of common tasks from which all other tasks can be obtained. In this perfectly monotonous environment, the best subset could be ascertained because the relative value of each task is accurately known and unchanging. Unusual sub-systems might be advantageously employed to perform tasks such as square root extraction or digital to analog conversion. Common sub-systems, like the arithmetic unit, might be specialized beyond recognition.

Obvious speed and cost gains are offered by such a design strategy, if it can be followed. The difficulty lies in the assumptions of the monotonous environment. In reality, everything important is not known about all tasks and the task environment is usually fraught with variables. Over a sufficiently long period of time, any computer facility is likely to encounter at least five uncertainties.

1
The domain of the task environment is imperfectly known at design time, and is in a state of flux thereafter. In order to accommodate new tasks, a computer facility can be expected to continually offer new services and a greater number of computations per day.
2
Neither the order of occurrence nor frequency of each task is exactly known and invariant.
3
The nature of any task is subject to revision as the task itself becomes better understood or improved upon.
4
Improvements develop in the equipment and methods for performing the tasks.
5
Opinions vary on the criteria for judging how well a system conforms to the task environment.

When the list is read through the eyes of the computer manufacturer, the uncertainties increase with the number of customers. No manufacturer can satisfy a large market with a single invariant computer system. Some "generality" is necessary in order for a system to accommodate the unpredictable. The problem is how much generality and in what form.

The word "generality" is used here to suggest "applicability to the task environment". As one crude indicator, a sub-system is considered to be more general than another if it is able to participate in the

performance of more tasks. An alternative indicator might be the containment relation. In typical arithmetic units, for example, the multiplication sub-system makes use of the adder sub-system. The adder could be considered to be the more general of the two because multiplication can be obtained from the adder without the existence of a special multiplication sub-system. The more pragmatic approach is preferred here. Otherwise, difficulty would arise in the comparison of unrelated sub-systems such as an adder and an input/output channel. In our terms, the adder is considered to be more general than the multiplier because it participates in at least as many tasks.

The penalty for excessive generality can be related to the match between sub-system generality and task environment. A sub-system is "miss-matched" to a task environment if it is able to perform tasks which never occur. A penalty is incurred if the substitution of a well-matched sub-system would reduce operating time or cost. In reality, of course, many factors influence the feasibility of such a substitution. For example, a system can be conceived which employs a small general purpose computer for the sole purpose of controlling a set of input/output channels. Being mass produced, the small computer might cost less than a channel controller designed and constructed specifically for the system. The penalty for the miss-match is fictitious unless some means are found for equating production costs.

The problem at issue here can now be restated. A form of generality is required which avoids miss-match penalties in an uncertain or changing task environment. An obvious solution is to allow system changes after design time. Each system is then potentially as general as it is variable.

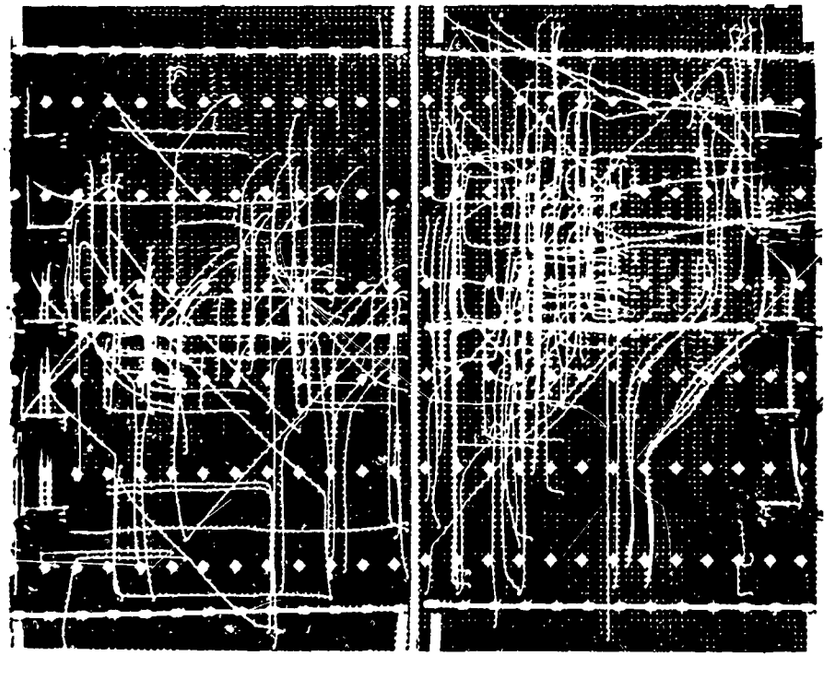
Manufacturers have long used variability to achieve generality. Options in computer organizations have been offered from the system configuration level to the opcode level. The union of all task environments of all customers can be thought of as the single task environment for which a manufacturer must design a monstrous computer system. An extensive product line can then be viewed as variability in the single monstrous system. Modularization and facility options are means for introducing variability at the sub-system level. This approach is clearly visible in the current IBM product line, billed as a single "system 360". The implication is that variations in a single highly general system are achieved by sub-system substitutions. Beyond the facility level, SDS offers the "program operator" by which program subroutines can be activated as if they were hardware opcodes. Customers who are able to afford hardware changes can replace the subroutines by sub-systems. Another approach to variability at the micro-level is the micro-program of IBM. A very fast "ROS" (read-only-store) memory can be rewired to execute non-standard opcodes. Unfortunately, this kind of micro-level variability is impractical for many customers. The money saved in system speed up must be balanced against the cost of engineering, parts, installation, and downtime during installation.

These organizational options are good; but variability is required even beyond the opcode and register level. Unless variability extends all the way down to the basic operational circuits, the variable unit may not be small enough to avoid miss-match penalties. For example, the "macro-modular" design concept extends variability to the character length register level. The idea here is to provide a diverse collection of interconnectable modules for which all engineering problems are presolved[1]. In order to provide freedom in the choice of interconnecting cable lengths, loading, heat dissipation, word length, and other design parameters, the modules contain synchronizing and standardizing circuits, which can significantly increase the time delays and costs of the task they perform. The task environment of any particular module may include these "overhead" tasks which would not be necessary if variability were possible at the basic circuital level. Miss-matches are possible because the overhead tasks do not necessarily occur in the task environment of interest. The penalties, in these cases, can be severe. Apparently, three guide lines are beneficial to the attempt to minimize miss-match penalties.

1
The generality of each sub-system should be initially well-matched to the expected task environment.
2
Variability should extend from the system level down to the basic operational circuit.
3
The production costs of implementing changes should be insignificant.

A model of the computer system can be developed which follows the three guide lines just given. In the model, the computer is considered to be decentralized into distinct sub-systems called "task control centers". Each center is well-matched to the domain of a specific task environment; but each is able to cooperate with other centers in the performance of tasks not entirely contained in any one domain. The task control centers act as intercommunicating building blocks from which an appropriate system for each task is constructed and dissolved by other centers or by programs.

A task is considered to be a sequence of events and a means for passing information. In hardware, the sequence of events correspond to successive states of sequential circuits. The information transmission devices are the interfacing circuits, such as read amplifiers and level shifters, which adapt the task to the real world.



A control hierarchy exists whereby the control of any center can be seized by some center of higher generality (if one exists). The effects of tasks performed at any level except the highest can be nullified so that the events of any task control center can always be controlled by an existing center of higher generality.

The hierarchy is achieved by means of appropriate lines of control and data communication. Each task center, except that of highest generality, has two properties.

1
At least one higher level center can supply and gain access to data used by the center.
2
At least one higher level center can initiate tasks which the center terminates. Synchronization can be achieved in this way, assuming that tasks terminated by the higher level center are "aborted" and can be nullified.

The assumption is made, of course, that the time required for control and data communication is small. Otherwise, the penalties for miss-match can be preferable to the cure.

Exceptions have been made for the highest level of generality. In our terms, the highest level is represented by the computer program. The task control center which corresponds to the program, in this hierarchy, is the "main memory" of the system. As suggested in the paragraphs concerning the "monotonous environment", an important function of main memory is to provide variability at the highest level of generality, the level at which all features of a task may be changed. (It is interesting to note that the alternative definition suggested, in the paragraph on generality, places main memory at the lowest level of generality.) The position of main memory as the apex of the hierarchy can be seen by reviewing the normal organization of computer systems. All tasks are initiated by opcodes emanating from main memory and terminated by task control centers such as the arithmetic unit. The

conventional "start" signal to memory can be viewed as the task termination signal from a lower level center. Finally, a communication path usually exists by which data and control can be passed between main memory and any task control center in the system. Because of its unique position in the hierarchy, main memory is the "safest" candidate for the position of higher level center when a new task center is designed. The hierarchy is certain to remain intact since main memory is certain to be of higher level generality than the new center.

One purpose of the hierarchy is to provide an "escape hatch" for emergencies arising from task environment uncertainty. Any task being performed with high efficiency by a center of low generality can also be performed at reduced efficiency by a center of higher generality, if one able to perform the task exists. The unexpected and unpredictable are accommodated by operations of low efficiency. The programs which specify these low efficiency operations are, in a sense, "simulators" of task control centers which do not yet exist. When a changing task environment ceases to change and becomes better understood, efficiency can be regained by the modification or addition of the appropriate centers.

Another purpose of the hierarchy is to provide a suitable structure for system development. The addition and modification of task control centers can be thought of as developmental rather than correctional, because the process begins during initial system design. As the system progresses before some task reality, reasonably detailed work is often in progress before some task control centers are specified. When units are finally available to the computer user, the level of specification and design known as "system configuration" is just in the beginning stages. Thus, redesign is not entirely different from initial design. The hierarchy is oriented towards a process which might be called "continuing design".

A system viewed in terms of this hierarchy appears to become least general at the fringes where highly specific demands are made by the real world. Task control centers at the human interfaces have few lower level centers beyond them in the hierarchy. They represent the lowest levels of generality. For example, a visual display system might provide a basic text editing and graphic manipulation task center at the display console. More elaborate tasks might be performed, under program control, by higher level centers able to override or nullify the effects of the editing and manipulation task center. However, those tasks which do fall within the domain of these centers can be performed at high efficiency.

One form of visual display control is typified by a DEC system in which a small general purpose computer, the PDP 8, acts as the task control center. The remainder of the computer system is accessible by means of a low capacity telephone line. The PDP 8 introduces more generality than necessary; but its low cost and high performance tend to reduce the miss-match penalties. The miss-match itself is reduced by a less general console such as MAGIC[2]. Some important tasks are performed by centers specifically designed for a visual display task environment. However, the MAGIC system runs the risk of being insufficiently general to accommodate unknowns and variables. The low capacity telephone line imposes severe time penalties on tasks performed jointly by the MAGIC console and the centers of higher generality. One of two situations can be expected. The match can be good, in which case the system will appear to be cumbersome or even inappropriate for some visual display tasks, and obsolescence will be relatively rapid. On the other hand, if the match is not good, generality has been designed into the console which might be better placed in the rest of the system. The cost of the generality cannot exceed the cost of higher capacity lines or the value of great distance between the console and the rest of the system. If it does, the argument can be made that the components which are allocated to the console for the purpose of achieving generality might be more efficiently employed at the central processor level where they can be shared by other task control centers.

The visual display consoles currently operating at Carnegie represent an attempt to avoid these time penalties[3]. Highly specific task control centers perform the same kinds of tasks that are performed by the MAGIC consoles. The cost of high capacity was made

reasonable by limiting the distance between the consoles and the main computer to 1000 feet. (The distance could have been greater but not miles greater.) Were the distance requirements significantly worse, the same problems would have arisen. The high communication capacity, along with certain other features, were designed into the consoles in an attempt to achieve a good match.

Other sub-systems of the G-21, the current computer system at Carnegie, are being designed or have been designed with the decentralization model in mind.

Some of these sub-systems, along with the visual display consoles, are represented in a conventional manner by Figure 1. The boxes are sub-system units and the lines are actual communication paths. This "communication network model" does not show the hierarchy of control explicitly. For example, the objects called "macro-modules" can exert control over the visual display consoles, although the model of Figure 1 implies an independence of the two sub-systems. The problem is that the form of control itself is implicit. It resides in the types of communication possible rather than in the existence of communication paths.

Figure 1

The Communication Network

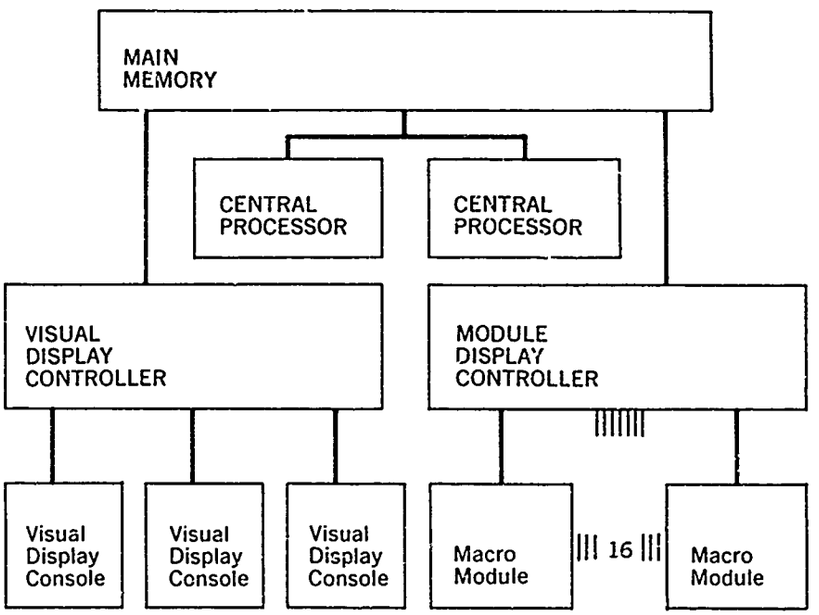
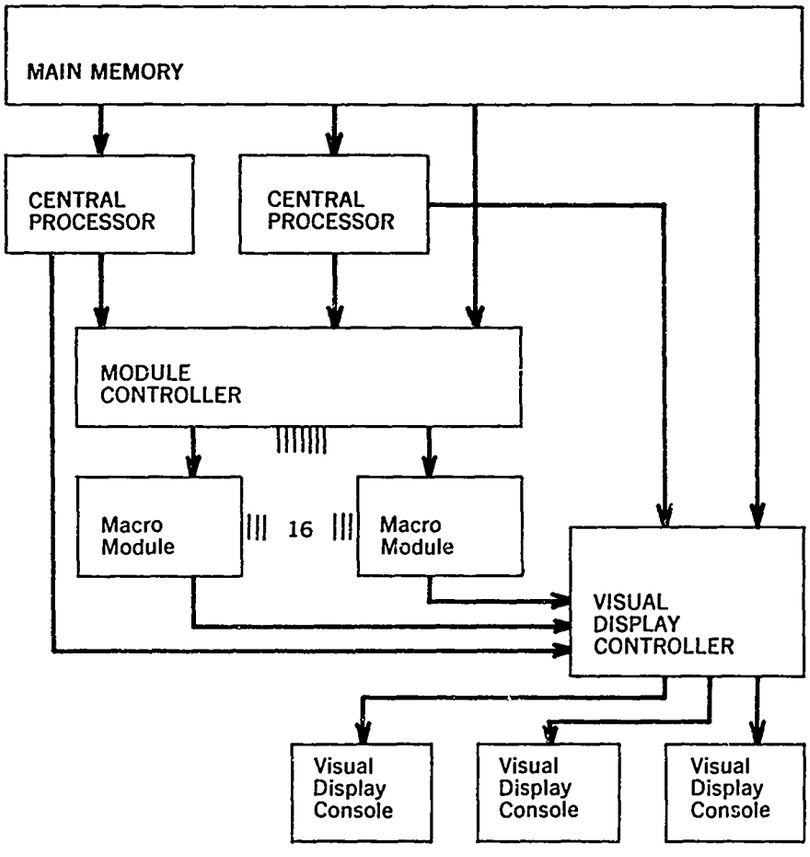


Figure 2 explicitly shows the hierarchy of control, which is identical with the hierarchy of generality. It corresponds to a "control model" of the system. Since the model need not emphasize natural physical boundaries, the factorization into distinct sub-systems is somewhat arbitrary. For example, the central processors can be thought of as a collection of sub-systems such as the arithmetic unit, the opcode decoding unit, and the interrupt processing unit. Beyond that, the arithmetic unit can be thought of as the adder, the registers, and the control circuits. The factorization chosen for Figure 2 emphasizes physical boundaries as an aid to comparison with Figure 1.

The main core memory serves as program memory for the central processors, the module controller, and the visual display controller. In addition, it serves as the regeneration memory for the visual display consoles. Typical tasks for the visual display controller are keyboard character encoding, character and line image formation, and format control. The task environment also includes specialized editing and manipulation tasks such as character insertion between displayed characters and figure or character string translation across the tube face. Tasks are specified by data which are stored in the main memory under control of the central processors and the module complex as well as

Figure 2

The Control Model



the visual display controller itself. Therefore, all tasks performed by the visual display sub-system can be nullified or modified by the other sub-systems which control memory information. Since these sub-systems have a higher level of generality, they are able to assist in the performance of visual display tasks or entirely replace them as the need arises.

The module complex is a means for introducing variability without incurring severe miss-match penalties. Briefly, the sixteen macro-modules are removable sub-systems interfaced to memory by the macro-module controller. Priorities, channel capacities, and memory locations can be assigned by programs stored in main memory. In this way, any macro-module can be given any share of the total resources. The tasks performed by the macro-modules are arbitrary and depend upon the needs of the moment. For example, a macro-module may be mounted which interfaces and samples an unusual input device such as a voice receiver. Another may convert standard memory words into "push-down" stacks. As needs change, a macro-module may be unplugged and replaced as if it were a reel of magnetic tape requested by the user.

The module complex is a way of following the second guide line given previously. Variability extends to the basic operational circuit because each macro-module can be designed and constructed at the micro-level. The module complex is organized in such a way that design parameters such as word length and channel capacity can differ for each macro-module. Within the macro-module itself, standard printed circuit cards can be substituted for each other or newly designed down to the integrated circuit. The decentralization model is an attempt to follow the first guide line. The third guide line requires further elaboration.

The continued construction of task control centers such as the macro-modules is practical only if the following conditions are met:

1
Labor and construction costs must compete favorably with the cost of programming a task and having it performed on more general but less efficient sub-systems.

2
The difficulties in designing, constructing, and debugging hardware tasks cannot be much greater, for the programmer, than they are in software tasks.

3
After the initial effort is invested by the designer (who is probably a programmer, not an engineer), it must be protected by low costs and reasonable ease in achieving redesign.

To summarize these conditions, hardware design can be neither overly expensive nor overly difficult if it is to be popular.

The cost of hardware variability, in terms of both expense and degree of difficulty, can be broken into three broad categories: material, installation time, and labor. Material costs have dropped radically, recently, with the advent of integrated circuit technology. Megacycle flip-flops, which formerly sold for $15 and up, are now available in single transistor cans for 32¢[4]. The reduced physical size offers further economic advantages. The costs of frames and chasis are not negligible. More important but less direct, smaller size means shorter wires. At the speeds common to contemporary computer circuits, shorter wires mean lighter circuit loading. Circuits can operate at lower power levels thereby reducing power supply costs as well as actual circuit costs. Miniturization also leads to labor costs. Since the entire circuit is contained within the transistor can, a great savings is achieved in the construction and stuffing of printed circuits.

Installation time falls into two categories: actual modification time and debugging time. The first category is virtually eliminated by the module complex. The macro-module itself is the only piece of hardware affected by modification, and it plugs in. Debugging time is reduced by a special debugging mode in which modules can be tested by diagnostic programs. Another technique, called STROBES, enables a technician to debug while the rest of the system continues in operation[5]. The continued operation of the system during macro-module installation is essential to reduce installation costs. A more important reduction can be obtained by means of an automated design program which includes an adequate simulator. Debugging can then be accomplished much more efficiently.

No such program is yet in existence at Carnegie. When it is, labor costs will also drop significantly. An adequate automated design program should permit the programmer who has no engineering skills to specify, "design", and debug macro-modules before construction. Until then, a mild form of automated design can be used. Forms can be completed by the programmer and turned over to a technician who has been taught to convert them into macro-module specifications. Next, the specifications can be transformed into macro-modules by a "wireman" of lower skill than the technician. The technician is then available to cooperate with the programmer in debugging. Hopefully, the programmer, the technician, and the wireman constitute a well-matched system.

The variability achieved by innovations such as the module complex provide a form of generality which avoids miss-match penalties. Miniturization technology has led to the drastic reduction in material costs which make variability at this level practical. Automated design programs promise reductions in labor time, installation time, and difficulty. At the same time, variability has become more important. This generation of computers is hard put to excel the last generation by a factor of ten. The previous step was on the order of 100 or more. If the trend continues, raw generality may not adequately absorb the apparently continual increase in needs. The high efficiency of specialization may be necessary.

A few overtures in this direction have been mentioned; but more grandiose examples are available. The P-5000 can be thought of as a mild specialization to the task of algebraic compilation. The SOLOMON is much more severe specialization to the task of array processing. However, raw specialization cannot be viewed as the panacea either. Few task environments are perfectly monotonous, especially that of the manufacturer. Generality is necessary if a system is required to accommodate variation.

References

1. Clark, Wesley A., Stucki, J. Mishell, Ornstein and M. Severo, "A Macromodular Approach to Computer Design, A Preliminary Report," Computer Research Laboratory, Washington University, St. Louis, Missouri, (1966).

2. Rippy, D. E., D. E. Humphries and J. A. Cunningham, "MAGIC: A Machine for Automated Graphics Interface to a Computer," Proc. FJCC, AFIPS (27) part 1, 819 (1965).

3. Quatse, Jesse T., "A Visual Display System Suitable for Time-Shared Use," Computation Center, Carnegie Institute of Technology, Pittsburgh, Pennsylvania (1965).

4. "Suggested O.E.M. Prices," Fairchild Semiconductor, Mountain View, California.

5. Quatse, Jesse T., "STROBES: Shared-Time Repair of Big Electronic Systems," Proc. FJCC, AFIPS (27) part 1, 1065 (1965).

Listing of Faculty

Saul Amarel

Visiting Associate Professor of Computer Science

B.S.	Technion, Israel (1948)
Ligenieur EE	Technion, Israel (1949)
M.S.	Columbia University (1953)

Chester Gordon Bell

Associate Professor of Electrical Engineering and Computer Science

S.B.	Massachusetts Institute of Technology (1956)
S.M.	Massachusetts Institute of Technology (1957)

Alan Hale Bond

Research Scientist in Computer Science

B.A.	Magdalen College, Oxford (1961)
Ph.D.	Imperial College, London (1966)

David C. Cooper

Associate Professor of Computer Science

B.Sc.	University College, London (1952)
Ph.D.	University College, London (1955)

David M. Dahm

Visiting Associate of the Computation Center

B.A.	Rice University (1958)
Ph.D.	Princeton University (1962)

James Eve

Visiting Assistant Professor of Computer Science

B.S.	Durham University (1954)
Ph.D.	Durham University (1957)

Robert W. Floyd

Associate Professor of Computer Science

B.A.	University of Chicago (1953)
B.S.	University of Chicago (1958)

Abraham Ginzburg

Visiting Associate Professor of Mathematics and Computer Science

B.Sc.	Technion, Israel Institute of Technology (1953)
M.Sc.	Technion, Israel Institute of Technology (1954)
D.Sc.	Technion, Israel Institute of Technology (1959)

Hellmut Golde

Visiting Associate Professor of Computer Science

Diplom-Ingenieur	Technische Hochschule (1953)
M.S.	Stanford University (1955)
Ph.D.	Stanford University (1959)

Robert A. Hopgood

Visiting Research Scientist of Computer Science

B.A.	Cambridge (1959)
M.A.	Cambridge (1962)

Sheldon Klein

Assistant Professor of Linguistics and Computer Science

B.A.	University of California at Berkeley (1956)
Ph.D.	University of California at Berkeley (1963)

Allen Newell

Institute Professor of Systems and Communication Sciences

B.S.	Stanford University (1949)
Ph.D.	Carnegie Institute of Technology (1957)

David L. Parnas

Assistant Professor of Computer Science and Electrical Engineering

B.S.	Carnegie Institute of Technology (1961)
M.S.	Carnegie Institute of Technology (1964)
Ph.D.	Carnegie Institute of Technology (1965)

Alan J. Perlis

Head, Department of Computer Science; Professor of Mathematics

B.S.	Carnegie Institute of Technology (1947)
M.S.	Massachusetts Institute of Technology (1949)
Ph.D.	Massachusetts Institute of Technology (1950)

Herbert Alexander Simon

Professor of Industrial Administration and Psychology; Associate Dean of the Graduate School of Industrial Administration

Ph.D.	University of Chicago
LL.D. (Hon)	University of Chicago
D.Sc. (Hon)	Yale University
D.Sc. (Hon)	Case Institute of Technology

John C. Strauss

Assistant Professor of Computer Science and Electrical Engineering

B.S.	University of Wisconsin (1959)
M.S.	University of Pittsburgh (1962)
Ph.D.	Carnegie Institute of Technology (1965)

Listing of Graduate Students

Babich, Alan
B.S. Carnegie Institute of Technology (1964)
M.S. Carnegie Institute of Technology (1965)

Balzer, Robert
B.S. Carnegie Institute of Technology (1964)
M.S. Carnegie Institute of Technology (1965)
Ph.D. Carnegie Institute of Technology (1965)

Bajzek, Thomas
B.S. Carnegie Institute of Technology (1966)

Berglass, Gilbert
B.S. Rensselaer Polytechnical Institute
M.S. Carnegie Institute of Technology

Berman, Victor
B.S.E. Cooper Union (1964)
M.S. Carnegie Institute of Technology (1965)

Broste, Nels
B.S. Carnegie Institute of Technology (1963)
M.S. Carnegie Institute of Technology (1964)

Caviness, Bobby F.
B.S. University of North Carolina
M.S. Carnegie Institute of Technology

Coles, L. Stephen
B.S. Rensselaer Polytechnical Institute
M.S. Carnegie Institute of Technology (1964)

Cunningham, Thomas
B.S. Carnegie Institute of Technology (1961)
M.S. Carnegie Institute of Technology (1962)

Davis, J. Roy
B.S. University of Texas (1957)
M.A. University of Texas (1962)

Darringer, John
B.S. Carnegie Institute of Technology (1964)
M.S. Carnegie Institute of Technology (1965)

Darius, Irani
B.S. University of Bombay (1963)
M.S.E.E. New York University (1965)

Earley, Jay
B.S. Carnegie Institute of Technology (1966)

Evans, Arthur
B.S. Carnegie Institute of Technology (1957)
M.S. Carnegie Institute of Technology (1959)
Ph.D. Carnegie Institute of Technology (1966)

Ernst, George
B.S. Carnegie Institute of Technology (1961)
M.S. Carnegie Institute of Technology (1962)
Ph.D. Carnegie Institute of Technology (1966)

Freeman, Peter
B.A. Rice Institute of Technology (1963)
M.A. University of Texas (1965)

Fikes, Richard
B.A. University of Texas (1963)
M.A. University of Texas (1965)

Fuller, Edmund L.
B.S. Iowa State University (1963)
M.S. Iowa State University (1965)

Gibbons, Gregory
B.A. University of California (1963)

Haney, Frederick
B.A. Ohio Wesleyan University (1963)
M.S. Colorado State University (1965)

Hansen, Gilbert
B.S. Case Institute of Technology (1962)
M.S. Case Institute of Technology (1964)

Iturriaga, Renato Fisico
B.A. University of Mexico (1963)
M.A. Carnegie Institute of Technology (1964)

Jeans, Christopher
B.S. Case Institute of Technology (1965)

Kane, Maureen
B.A. Ladycliff College (1953)
M.A. Fordham University (1954)

King, James C.
B.A. Washington State University (1962)
M.A. Washington State University (1964)

Krack, John L.
B.S. Case Institute of Technology (1966)

Krutar, Rudolph
B.S. Carnegie Institute of Technology (1966)

Lansac-Fatte
License Faculte de Science (1962)
Ingevic ENSECHT

Lauer, Hugh
B.S. Antioch College (1965)

Lieman, Stephen
B.A. Hunter College of the City University of N.Y. (1964)

LeQuesne, Peter
B.S. University of British Columbia (1963)

Lilly, Gordon
B.Sc. University of Manchester, London (1948)
M.S. Carnegie Institute of Technology (1960)
M.Lit. University of Pittsburgh (1963)

Lindstrom, Gary
B.S. Carnegie Institute of Technology (1965)
M.S. Carnegie Institute of Technology (1965)

Mayer, Bernard
B.S. Carnegie Institute of Technology (1961)
M.S. Carnegie Institute of Technology (1965)

McCreight, Edward
B.A. College of Wooster (1966)

Manna, Zohar
B.A. Israel Institute of Technology (1961)
M.A. Israel Institute of Technology (1965)

Mitchell, James
B.S. University of Waterloo (1966)

Modesitt, Kenneth
B.S. University of Illinois (1963)
M.S. Stanford University (1965)

Moore, James, Jr.
S.B. Massachusetts Institute of Technology (1964)

Oliver, C. Frank
B.S. Carnegie Institute of Technology (1965)

Parnas, David
B.S. Carnegie Institute of Technology (1961)
M.S. Carnegie Institute of Technology (1964)
Ph.D. Carnegie Institute of Technology (1965)

Purcell, Gerald
A.B. Whitman College (1960)
M.A. Howard University (1963)

Ramey, Robert
Ph.B. Northwestern University (1954)
M.S. Northwestern University (1956)
M.A. Northwestern University (1957)

Ross, Daniel
B.S. California Institute of Technology (1960)
M.S. California Institute of Technology (1961)

Rubin, Ira
B.S. Pennsylvania State University
M.S. Carnegie Institute of Technology

Siklossy, Laurent
B.A. Yale University (1963)
M.A. Harvard University (1964)

Standish, Thomas
B.S. Yale University (1962)

Thompson, Carol
B.S. Carnegie Institute of Technology (1964)
M.S. Carnegie Institute of Technology (1965)

Taranto, Donald
B.S. City College of New York (1953)
M.S. Adelphia University (1959)

Wagner, Robert
B.S. Massachusetts Institute of Technology (1962)
B.A. Columbia University (1964)

Waldinger, Richard
B.A. Columbia College (1964)

Listing of Staff Administrators

David H. Nickerson	Director of the Computation Center
	Polly A. Breza, Supervisor of 360 Programming Task Group
	Janet Delany, Research Programmer
	Janet Fierst, Supervisor of Language Development
	Albin L. Vareha, Jr., Programming Supervisor
	Cynthia Yang, Research Supervisor
	Harold Van Zoeren, Research Programmer
Charles Pfefferkorn	Manager of Planning
	Melvin Boksenbaum, Planning Assistant
	Janet Wurmb, Technical Writer
	Tom Cunningham, Supervisor of User Consultants
	Pat Pringle, Administrative Assistant
Carl Lefkowitz	Manager of Operations
	Roy Weil, Supervisor of System Maintenance and Testing
	Queen Purcell, Supervisor of Operations
	Nicholas Kydon, Supervisor of Use Analysis
	Art Yaffe, Supervisor of Operations
Jesse T. Quatse	Manager of Engineering Development
	Manuel Langtry, Supervisor of Engineering Development

Listing of Publications

Klein, Sheldon, "Automatic Paraphrasing in Essay Format," *Mechanical Translation* (8) (1965).

Klein, Sheldon, "Control of Style With a Generative Grammar," *Language* (41) No. 4 (1965).

Newell, Allen, "Limitation of the Current Stock of Ideas for Problem Solving," *Conference on Electronic Information Handling*, 195-208 (1965).

Newell, Allen, "On Protocol Analysis," Computer Science Department Colloquium, University of Wisconsin, December (1965).

Newell, Allen, "Programs As Theories of Higher Mental Processes," *Computers in Biomedical Research* (2) 141-172 (1965).

Newell, Allen and Herbert A. Simon, "An Example of Human Chess Play in Light of Chess Playing Programs." Reprinted from Norbert Weiner and J. P. Schade (eds) *Progress in Biocybernatics* (2) Elsevier Publishing Company, Amsterdam (1965).

Newell, Allen and Herbert A. Simon, "Heuristic Problem Solving By Computer," Margo A. Sass and William D. Wilkinson (eds) *Computer Augmentation of Human Reasoning*, Chapter 3, 25-36 (1965).

Newell, Allen and Herbert A. Simon, "Simulation of Human Processing of Information," reprinted from the *American Mathematical Monthly* (72) No. 2, Part II (1965).

Newell, Allen and George Ernest, "The Search for Generality," *Proceedings of IFIPS Congress*, New York (1965).

Newell, Allen, P. Keller, and F. Tonge, "Quikscript," *Communications of the Association for Computing Machinery* (8) no. 6 (1965).

Perlis, Alan J., "Construction of Programming Systems Using Remote Editing Facilities," Presented at IFIP Congress, May (1965).

Quatse, Jesse T., "Strobes," presented at Fall Joint Computer Conference, November (1965).

Strauss, J. C., and A. Lavi, "Parameter Identification in Continuous Dynamic Systems," *1965 IEEE International Convention Record*, Part VI, IEEE, 49-61, March (1965).

Strauss, J. C., and W. L. Gilbert, "SCADS: A Programming System for the Simulation of Combined Analog Digital Systems," *Simulation Languages*, supplement to the Proceeding of the 1965 JACC, June (1965).

Bond, Alan Hale, "A Systematic Study of the Baiags Bootstrap Method," *Physical Review*, June 24 (1966).

Cooper, David C., "The Equivalence of Certain Computations," *The Computer Journal* 9 (4) 45-52 (1966).

Cooper, David, "Mathematical Proofs About Computer Programs," *Machine Intelligence I*, D. Miche (ed) Oliver and Boyd, to be published November (1966).

Cooper, David V., "Reduction of Programs to a Standard Form by Graph Transformations," *Proceedings of International Seminar on Graph Theory and Its Applications*, Rome, Italy, July (1966) to be published January (1967).

Cooper, David C., "Some Transformations and Standard Forms of Graphs with Applications to Computer Programs," *Machine Intelligence II*, D. Miche (ed) Oliver and Boyd, to be published June (1967).

Cooper, David C., "Theorem Proving in Computers," *Advances in Programming and Non-numerical Computation*, L. Fox (ed) Pergamon Press 155-182 (1966).

Eve, James, G. A. Baker, H. E. Gilbert, and G. I. Rushbrooke, "On the Heisenberg Spin 1/2 Ferromagnetic Models," *Physical Letters 20*, 146-147 (1966).

Floyd, Robert W., "Assigning Meanings to Programs," AMS Symposium on Applied Mathematics, April 6 (1966).

Floyd, Robert W., "Edited Proceedings of Symposium on Symbolic and Algebraic Manipulation," Washington, D. C., March (1966).

Floyd, Robert W., "The Verifying Compiler: An Approach to Rigorous Debugging," Seminar, IBM, from the New York Programming Center to IBM Poughkeepsie and IBM Kingston, Tuesday, April 5 (1966).

Ginzburg , Abraham, "Homomorphic Images of Graphs and Some Applications to Automata Theory," Math Colloquium, SUNY at Buffalo, February (1966).

Ginzburg , Abraham, "Some Problems of Automata Theory," Math Colloquium, Carnegie Institute of Technology, Pittsburgh, Pennsylvania, April (1966).

Hopgood, F. R. A., J. Hubbard, and D. E. Rimmer, "Weak Covalency in Transition Metal Salts," *Proceedings of the Physical Society 88*, 13-36 (1966).

Iturriaga, Renato, T. A. Standish, R. A. Krutar, and J. C. Earley, "Techniques and Advantages of Using the Formal Compiler Writing System FSL to Implement a Formula ALGOL Compiler," *Proceedings of the SJCC*, 241-252 (1966).

Klein, Sheldon, S. Lieman, and G. Lindstrom, Diseminer: A Distributional-Semantics Inference Maker, Carnegie Institute of Technology, Pittsburgh, Pennsylvania, June (1966).

Newell, Allen, Discussion of Papers by Dr. Gagne and Dr. Hayes, *Problem Solving: Research, Method, and Theory*, Benjamin Kleinmuntz (ed) Wiley, 171-182 (1966).
Newell, Allen, "Four Lectures on Artificial Intelligence and Information Processing in Psychology," Stanford University, March (1966).

Newell, Allen, "How Humans Solve Problems," Talk, Robert Morris Junior College, to students and interested faculty, January (1966).

Newell, Allen and Herbert A. Simon, "Information Processing in Computer and Man," *Science in Progress*. Wallace R. Brode (ed) Yale University Press, 333-362 (1966).

Parnas, David L., "A Language for Describing the Function of Synchronous Systems," *Communications of the Association for Computing Machinery* (9) no. 2 (1966).

Parnas, David L., "On Facilitating Parallel and Multi-Processing in ALGOL," *Communications of the Association for Computing Machinery* (9) no. 4 (1966).

Parnas, David L., "On the Preliminary Report of C^{3}S," *Communications of the Association for Computing Machinery* (9) no. 4 (1966).

Parnas, David L., "On the Use of the Computer in Engineering Education Without a Programming Prerequisite," *Journal of Engineering Education* (56) no. 8 (1966).

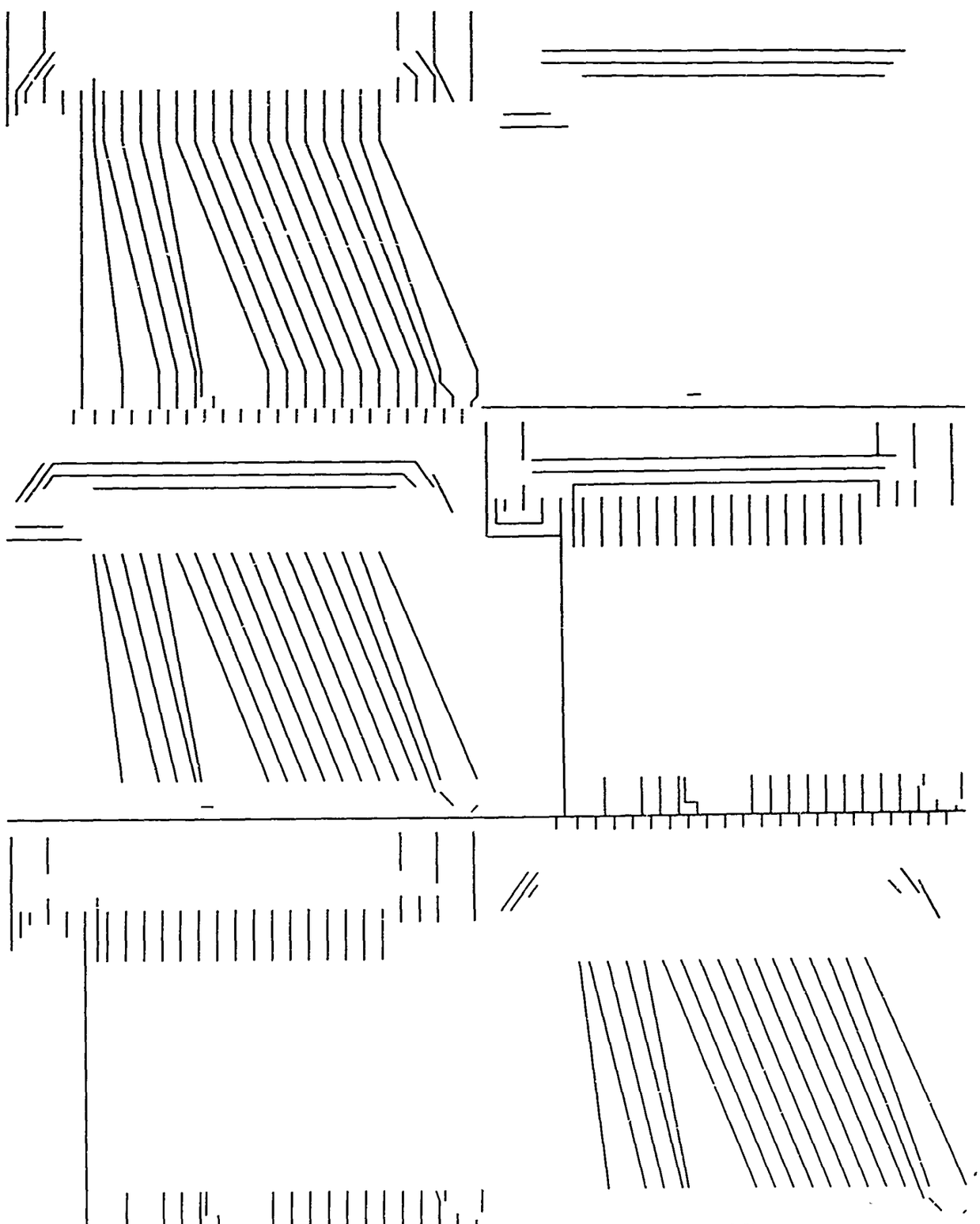
Parnas, David L., "State Table Analysis of Programs in an ALGOL-Like Language," *Proceedings of 21st National Conference, Association for Computing Machinery* (1966).

Perlis, Alan J., "A. M. Turing Lecturer for 1966."

Perlis, Alan J., R. Iturriaga, T. Standish, SICSAM Symposium on Symbolic and Algebraic Manipulation, "Definition of Formula ALGOL," Washington, D. C. March 29 through March 31 (1966).

Quatse, Jesse T., "Visual Display System Suitable for Time-Sharing," Seminar, IBM, Endicott Laboratory, March 30 (1966).

Strauss, J. C., "Optimal Tracking of Nonlinear Dynamic Systems," *Symposium on Optimization Techniques*, A. Lavi and T. Vogl (eds) J. Wiley and Sons (1966).



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)

Computation Center and Department of Computer Science
Carnegie Institute of Technology
Pittsburgh, Pennsylvania

2a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED

2b. GROUP

3. REPORT TITLE

Computer Science Research Review

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Scientific Final

5. AUTHOR(S) (First name, middle initial, last name)

Edited by Joyce Nissenson

6. REPORT DATE: 1966

7a. TOTAL NO. OF PAGES

7b. NO. OF REFS

8a. CONTRACT OR GRANT NO.: SD 146

b. PROJECT NO.

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10. DISTRIBUTION STATEMENT

Unlimited

Defense contractors may obtain copies from:
Defense Documentation Center, Defense Supply Agency
Cameron Station, Alexandria, Virginia 22314

11. SUPPLEMENTARY NOTES

None

12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

ANNUAL REPORT

Table of Contents

DD FORM 1 NOV 68 1473

Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT