THE PRODUCTION OF BETTER MATHEMATICAL SOFTWARE

by

LLOYD D. FOSDICK
Department of Computer Science
University of Colorado
Boulder, Colorado 80302

Report #CU-CS-001-72

March 1972

Work supported in part by NSF Grant GJ-660

THE PRODUCTION OF BETTER MATHEMATICAL SOFTWARE

Lloyd D. Fosdick
Department of Computer Science, University of Colorado,
Boulder, Colorado, 80302

ABSTRACT

Some observations are made on steps to be taken toward the creation of better mathematical software. These steps suggest the need for a coordinated effort and the creation of a center to focus activities in this area.

KEY WORDS AND PHRASES mathematical software, programming.

CR CATEGORIES 4.0

INTRODUCTION

Mathematical software could and should be considerably better than it is. Some suggestions are made here for work aimed at improving the present situation. These concern the areas of documentation, standards, validation, and design. A major theme is that there is a lot of overlap between these areas and if substantial progress is to be made, a well-coordinated effort is necessary. A patchwork of good but inconsistent software is not a desirable situation. To achieve this coordination, a center devoted to work in this area is proposed.

A careful look at the computer literature shows that the subjects of machine design, numerical analysis, formal languages, and automata theory have received a good deal more attention as objects of serious study than computer algorithms. Another, far more direct, way of observing the lack of attention to this area is by looking at the software library for most computer centers and, further, considering the money and manpower devoted to it compared with other things the center does. Even in leading scientific laboratories mathematical software simply does not receive the attention

it should. However, there is evidence that this situation is changing. More good algorithms are being published, the subject of validation and testing computer algorithms has received increased attention, and at least one company, IMSL, is making a serious attempt to market high-quality mathematical software. This is partly a result of a recognition of the high cost of poor programs; ones with bugs which cause them to fail unexpectedly, ones that are so poorly documented that it is impossible to understand how they are to be used, and ones that depend very strongly and obscurely on some particular system so that it is a major undertaking to move them to another system. There is also a growing recognition of the fact that we must understand how to build very large programs if we are to effectively utilize future computers. Now it is a major effort to put together a program of 1,000 to 5,000 statements; we can anticipate having to deal with programs several orders of magnitude larger than this in the future.

Some of the software problems seem to be rather dull intellectually: preparation of good documentation and design of standards does not excite the interest of most computer scientists. However, there are some intellectually stimulating problems even in these areas as will be set forth below. The design, analysis, and determination of correctness seem to be more challenging subjects than documentation and standards; problems and suggestions for developments in these areas are also set forth below.

The software problems we face are related one to another in obvious ways. Users of computer software see the problems differently, of course, than the producers who often would prefer not to be bothered by the practical needs of users. Users would like to have a collection which is

reliable, extensive, internally consistent, and easy to use. Meeting this need requires a large coordinated effort involving work in all of the areas cited above. Below there is a suggestion for a national software center to help meet this need.

DOCUMENTATION

Most of us are familiar with the conventional documentation accompanying a program and its inadequacies. Various guidelines for the preparation of documentation have been produced, most appearing as internal reports, but some have been published (1,2). Some of the algorithms published in the Algorithms Department of the Communications have particularly good documentation, for example algorithms 343 (3) and 395 (4). Other examples of good documentation can be found in the collection of linear algebra routines being prepared under the NATS project (5).* Rather than dwell on the conventional aspects of this subject, let us look at some alternatives that might be useful. First let us recognize that the two kinds of information a user might need relative to a program are distinguished in the following way: one kind can be deduced from the program itself and the environment in which it will operate, for example the method of calling the program; the other kind cannot be deduced this way, for example the fact that the program is based on an algorithm designed by I. Smith and reported in a particular journal. One ought to try to obtain automatically as much information of the first kind as possible. However, beyond the automatic production of flow diagrams, which has received quite a lot of attention (6,7,8,9), little else seems to have been done in this direction. A far more serious effort could and should be made in extracting information automatically from a program: the possibility of extracting information

*Footnote on page 3a.

"This project is aimed at the production and distribution of high-quality mathematical software in selected areas. It is a collaborative effort involving primarily Argonne National Laboratory, Stanford University, and the University of Texas. A number of universities and laboratories in the U.S. and elsewhere are serving as test sites for the programs produced by this effort.

concerning acceptable bounds on input and output parameters, execution time, and error conditions needs to be explored. Automatic determination of strict bounds on program parameters is probably impractical or impossible in most cases, but useful bounds may be obtainable. Automatic determination of information on execution time is not meant to imply automatic determination of convergence rates for iterative processes; in such cases, however, it should be possible to obtain an execution time for each cycle of the iteration.

Different users often require different information about a program and for this reason it seems useful to attempt to supply information about a program through a question-answer process at a terminal. Under such a system a minimal amount of information, that which would serve to identify the program and be of general interest to all potential users, would be distributed in the conventional way; other information would be obtained from queries at a terminal. A system of this type properly designed could allow an individual to ask very detailed questions about a program, for example, questions about frequency of execution of parts of the program when selected inputs are applied; execution paths; storage references; etc. Combined with an editing facility, a user could make alterations to the program to suit his particular needs.

Such a system could also be used by designers of the documentation in collecting and editing information of the second kind. For example, a set of queries generated by the computer, such as -- Author's name? Last revision date? References? -- etc., and appropriate responses by a human at a terminal would force the documentation into a particular format, thus making it more consistent. Records could be kept on the queries

submitted at terminals by users and this data could be used to expose problems with the documentation.

It is apparent that the design of a system of this type would require a major effort and to justify the cost of production, it would have to be widely used. It would not be static, so a continuing maintenance effort would be necessary.

STANDARDS

One of the biggest problems associated with the distribution and exchange of software is the lack of standards for hardware and software. For example, transmission of a program from one installation to another by the medium of magnetic tape is often unduly complicated because of hardware or software (or both) mismatches. For this reason, and many others, the lack of standards causes the waste of a lot of time and money, as well as impeding progress. Efforts to resolve problems concerned with hardware and software standards are being made by the American National Standards Institute Sectional Committee X3. Reports of subcommittees of the X3 Committee are published in various issues of the Communications. A review of work in this area to 1967 can be found in a paper by Steel (10).

Standards are not very effective if they are not used. At the moment there does not appear to be a very effective way to force manufacturers to conform to standards, especially those which insure compatibility of their products with one another. We can reasonably anticipate that this problem will always be with us, but its intensity could be greatly reduced. Such a reduction could only be imposed by a very large purchaser of computer hardware and software, that is the federal government, but whether or not effective progress will be made here is far from clear.

Congressman Jack Brooks has expressed some of his feelings in a letter reprinted in this journal (11). The problem is politically charged. Furthermore failure to meet standards, although wasteful in human and material resources, has not had dramatic direct effects: planes do not crash, people do not become critically ill, and so forth.

There is an activity in standards directly related in mathematical software which could and should be actively pursued. This is the production of tables of values of elementary and higher functions in machine readable form. In checking a numerical algorithm, say to produce Bessel functions of a certain type, one is forced to go to a book of tables to verify the numbers. The uncertainty of this approach is apparent. One could argue that such tables are unnecessary because very carefully checked standard programs could be used instead. Indeed a program may be viewed as an encoding of its output. The difficulty here is that the output of programs is dependent on a machine and its associated software system and this evironment is not constant. Perhaps it will be possible to have programs for which the output can be guaranteed to be independent of the environment; if so, it will not be soon. There have been some isolated efforts to produce tables on magnetic tape, one by a group at the National Bureau of Standards (12), and another in connection with the NATS project (3). A far more extensive effort is necessary. Success requires such a project to be well coordinated to insure consistency of the material. A cooperative effort like that which produced the Handbook of Mathematical Functions (13) would be appropriate.

There are many issues associated with the production of standard tables which need to be resolved: precision of the tabulation, which arguments, mode of recording (e.g., binary, decimal), and so forth. It seems

reasonable that the equivalent of about fifty decimal places would be an adequate precision and that binary rather than decimal recording of information would be appropriate. The choice of arguments would be expected to depend on the variation of the function and thus might be connected with the modulus of continuity; also certain arguments are of particular interest, for example arguments corresponding to extrema and zeros. These issues and related ones for the production of such tapes need careful study and agreement among potential users should be sought.

VALIDATION

Users of computer programs need reasonable assurance of reliability. A lot of research is being done in the area of proving correctness of programs: some of the best known work in this field has been done by McCarthy (14), McCarthy and Painter (15), Floyd (16), Hoare (17), Dijkstra (18), and Naur (19,20). A bibliography of work in this area is given by London (21,22). Most of this work is rather theoretical and although of considerable importance still leaves us some distance from the practical goal of validating, say, a program to solve differential equations. Hull's work (23) on the validation of mathematical software is an exception: he is attacking directly the problem of validating programs in the area of linear algebra and differential equations.

Certain simple measures could be taken for all programs, especially those to be used by a wide community, which would help to establish reliability. One is to guarantee that every statement in the program has been executed at least once during the course of tests applied to it. Providing for this kind of information is straightforward, it being only necessary to identify all branches in the program and insert a command

which will set an appropriate flag. A program which does this for Fortran codes, actually counters rather than flags are used, has been discussed in (24). For short programs such flag setting statements are easily inserted by hand but such a process is prone to human error. In spite of the obviousness of this kind of minimal check, and Wegstein's comment on this (25), not one manuscript in more than two hundred received by the Algorithms Department of the Communications in over two years indicated that a test of this type had been made.

For some simple program structures, the fact that every statement has been executed at least once during the tests implies that every path through the program has been traversed. Figure 1 illustrates such a case. Figure 2 illustrates another provided that we do not distinguish between one or more traversals of a path. It should be apparent that tree structures have this property. On the other hand, a simple structure like that shown in Figure 3 does not have this property: it is evident that traversal of paths (a), (b), and (c) would set all flags.

Another verification procedure of this type identifies traversal of branch pairs. Thus in Figure 3 path (a) traverses pairs [1,3] and [3,4]; path (b) traverses pairs [2,3] and [3,5]; etc. These traversals can be identified by a transition matrix as shown in Figure 4 where each entry is 0 or 1 with a 1 in the [i,j] position representing traversal of the branch pair [i,j]. It is evident that a verification procedure based on the identification of pair traversals would not generally verify that all paths through a structure had been traversed. The pair traversals generated by paths (a), (b), and (c) of Figure 3 would yield the same transition matrix as shown in Figure 4. Notice, however, that the simple transformation indicated in Figure 5 on this structure will produce one in which traversal

of all branch pairs guarantees traversal of all paths. This transformation is suggested by the obvious fact that if branches 1 and 4 in Figure 3 have been traversed, then branch 3 must have been traversed.

It seems evident that these ideas can be extended in various ways and that they can be exploited for the purpose of checking computer programs. One would like to be able to obtain global information, such as the fact that all paths have been traversed in a testing procedure, from local information: branch pair traversals, and so forth. The extent to which an approach of this type can be successful in practice depends on the structure of real programs, but unfortunately very little is known about this. One effort in this direction by Knuth was reported recently (26); this was concerned primarily with the frequency of use of Fortran statements and expressions. Kuck at Illinois has also been looking at the structure of real programs to explore more effective machine organizations for execution of these programs.

Another idea of importance to this area is incorporating a validation procedure within the design procedure of the algorithm itself. Dijkstra (18) has tried to exploit this idea in the construction of a programming system. A more formal approach is presented in a recent report by Floyd (27) in which an interactive scheme for designing a correct program is described; as an example he uses the design of a program to locate a symbol in a sorted table.

In the analysis of a program for the purpose of validation, it might simplify matters to distinguish statements which control the path of execution from those which do not. Wilkes (28) calls statements in the first group the outer syntax, and those in the second the inner syntax.

Thus statements such as

GO TO 55 (Fortran) or go to L ; (Algol)

and

DO 15 J=1,25 (Fortran) or for j:=1 step 1 until 25 do (Algol)

belong to the outer syntax, and statements such as

X = Y+5.0*Z (Fortran) or x:=y + 5.0 x z; (Algol)

belong to the inner syntax. This distinction permits the analysis of a program to be split into potentially more tractable pieces. However, statements such as

IF (X.LT.12.5) GO TO 15 (Fortran) or if x<12.5 then do (Algol)

introduce a coupling between the inner and outer syntax. Points in the program at which such statements appear are particularly critical points for diagnostic probes. We can anticipate that the difficulty in analyzing a program's behavior is connected with the number of such points.

In dealing with the inner syntax, the analysis of roundoff error arises. This analysis can be split into two parts. In one part the machine operations $\oplus$, $\ominus$, etc., are replaced by exact operations +, -, etc., according to formal rules such as

$$x \oplus y = (x + y)(1 + \rho)$$

where ρ is a parameter whose exact value is unknown but satisfies an inequality

$$|\rho| < \epsilon$$

where the number ϵ is known and depends on the roundoff error of the machine. With these rules and a few key theorems, one can convert expressions involving machine operations into expressions involving exact operations and terms linear in ρ, whose exact value is again unknown but is bounded as in the inequality above. This conversion is perfectly straightforward, though

exceedingly tedious, and could be done by a computer. The second part of the error analysis involves establishing upper bounds on intermediate results, using a variety of theorems, to arrive at an error bound for the final result of the computation. It is not now apparent how this second part could be made automatic, but certainly the labor of the total process could be simplified if at least the first part were turned over to a machine. For a posteriori error estimates the second part of the analysis could be dropped and instead the computer, during the course of the calculation, could record the extrema for the key intermediate results. Such a procedure would appear to require less work at execution time than interval arithmetic or significance arithmetic, and it has the attractive feature of including at least a portion of the a priori analysis.

DESIGN

New programs are constantly being constructed, but many of these are not new in any significant way. Too often the authors of these programs do not pay attention to the work of others. This may be a reflection on the authors; it is certainly a reflection on the mechanisms for communicating programs. I am not referring here so much to the publication of programs as to the ways we have of talking about programs, and of describing them to others for various purposes. It is essential to have a listing of the program for purposes of understanding it but usually more is needed. Our weakness in communicating programs is evident from the fact that many prefer to write their own program rather than to try to use and understand someone elses. Much that could and should be done in this area has been discussed above in connection with documentation. A related subject is portability, the ability to use a program in different

environments: different people, different software, different hardware.

There are simple, obvious things for programs written in the standard languages which would improve their portability. One is to put all machine-dependent parameters in one place, identify them as such, and give a prescription for changing them if the machine environment changes. Programs frequently have parameters which control storage allocation, execution time, and accuracy. Again these should be brought together, identified, and prescriptions given for changing them which might help a user willing to sacrifice one for the other, say speed for accuracy.

It should be possible to design some programs in such a way that the fundamental parameters of speed, storage, and accuracy might be adjusted automatically. No serious efforts in this direction seem to have been made. An obvious place where a scheme of this type seems practical is in function approximation. Let us assume that some function $f(x)$ is to be approximated by another $\bar{f}(x)$ with error $\epsilon(x)$; thus

$$f(x) = \bar{f}(x) + \epsilon(x).$$

For example $\bar{f}(x)$ might be a rational function or it might be a table, or both. Consider now a formal description of an algorithm for the evaluation of $\bar{f}(x)$ in which details related to a particular implementation are left unspecified; for example, if $\bar{f}(x)$ were a rational function, then the numerator and denominator polynomials would be left unspecified. This formal description of the algorithm we will call a meta-algorithm. A program to evaluate $\bar{f}(x)$ on a particular machine with certain conditions on speed, storage, and accuracy is to be obtained from the meta-algorithm as follows: A special kind of compiler, call it a mapper, would read the meta-algorithm along with data specifying conditions on speed, storage, and accuracy. The mapper would produce as output either a program to evaluate

$\bar{f}(x)$, meeting the specified conditions, or it would produce a message that it could not produce a program meeting the conditions. This ambitious scheme may only be partly realizable. It might be best done in an interactive mode so that a human could respond to unforeseen circumstances or circumstances that have no presently known algorithmic solution.

The development of new machine architecture, parallel processors, pipeline processors, and various storage organizations leads to the development of new algorithms which take advantage of the architecture. There should be more effort extended in the opposite direction; that is, finding the best architecture for the algorithms. We should in fact regard both architecture and algorithms as flexible in a search for the best combination for a class of problems. Here again, we are hampered by poor knowledge of programs in common use. It does not help a great deal to put a large parallel processor to work on a reactor calculation if only 30% of the calculation is concerned with solving differential equations on some grid by a finite difference scheme and the remaining 70% is concerned with messaging data in a way that does not exploit the efficiency of a parallel processor.

Turning attention now to the construction of specific kinds of algorithms, an examination of the CALGO index (29) shows obvious areas of weakness. For example, the list of algorithms for solving integral equations and differential equations is particularly short. Algorithms for the approximation of functions of more than one variable are almost non-existent. The algorithms for higher functions could be improved. In this connection there has been some discussion of the development of a collection of algorithms for the higher functions, analogous to and perhaps paralleling the Handbook of Functions (13).

SOFTWARE CENTER

A number of suggestions have been made here for future work in the software area, particularly mathematical software. These are divided into four subareas: documentation, standards, validation, and design. There is a strong coupling between these areas and any serious effort toward the production of good software cannot ignore this. Since the production of good software is difficult, expensive, and requires a sustained effort, it is appropriate to establish a center dedicated to this effort. Such an idea is not new: most recently Rice (30) has made such a suggestion; at an earlier time J. Schwartz had proposed a center for research in Computer Science. Rice has mentioned that software development is an activity of low professional status and thus good people are not attracted to it. This is true but there is another reason that good people are not attracted to this area. Scientists would like to believe or feel that their work will have some impact, presumably good. Unfortunately, the lack of coordination and standards in this area make it very likely that a person's efforts in the production of good software will go unrecognized and unused, except perhaps for his local enviornment. This discourages the best efforts of good people. A center which would serve as a focus for software activities could help to overcome this problem. For example, if one activity of the center was the development, maintenance, and dissemination of a library of mathematical software then there is the potential for wide recognition and utilization of the fruits of an individual's software efforts.

It does not seem that it should be necessary to point out the advantages of such a center to software users, but here are some. A central library for high-quality mathematical software could cut out an enormous duplication of effort taking place now in both private and public

installations. Resources now being wasted by duplication could be turned on the important problems of validation and documentation which are now so neglected. The center could be used to check software generated elsewhere to insure that standards of testing and compatibility have been met. Finally, there is the potentially enormous effect that such a center would have on all software merely by its leadership.

There are certain practical aspects of such a center that I feel are important. It should allow the easy exchange of information and people. Thus it must not be hampered in any way by the kind of proprietary secrecy associated with private concerns. Further, it must be charged with the kind of intellectual stimulation one finds at a university. No one can give a prescription to guarantee this but the chances that it will be there are greatly improved if substantial efforts are made to allow easy exchange of people. Perhaps 50% or more of the workers at such a center might be visitors with long, say two years, or short, say a month, appointments. Visitors would be from universities, government laboratories, and private industry. There is another reason for this policy. It is anticipated that the facilities at such a center would be far more extensive and powerful than those available elsewhere, thus visitors would have access to facilities not otherwise easily available to them. Of course, some of these facilities could and should be made available over a communication network. However, such networks are very expensive, and this will preclude going far beyond low data rate terminals, such as a teletype, for some time in a network with many remote stations.

It seems unlikely that funding for such a center would come from anywhere but the federal government. Perhaps some funding from private sources, say for fellowships and visiting positions, might be obtained.

There is the possibility that software produced by this center could be sold to recover some costs. Whether this could or should be done is unclear.

The proposed center is concerned with one area of computing, mathematical software. Other areas might be included or there might be a network of centers, each for a different area. The establishment of a mathematical software center seems like a reasonable first step.

REFERENCES

1. Howarth, R.J., and Lim, A.L. An approach to program documentation. Comp. Bull. 13 (August 1969), 291-295.

2. Walsh, D. A Guide for Software Documentation. Advanced Computer Techniques Corporation (1969).

3. Grad, J., and Brebner, M.A. Algorithm 343, Eigenvalues and eigenvectors of a real general matrix. Comm. ACM 11 (Dec. 1968), 820-826.

4. Hill, G.W. Algorithm 395, Student's t-distribution. Comm. ACM 13 (Oct. 1970), 617-619.

5. NATS Project. SIGNUM Newsletter 6, 3 (1971), 5.

6. Knuth, D. Computer drawn flowcharts. Comm. ACM 6 (1963), 555-563.

7. Sherman, P.M. Flowtrace, a computer program for flowcharting programs. Comm. ACM 9 (1966), 845-854.

8. Obrien, F., and Beckwith, R.C. A technique for computer flowchart generation. Comp J. 11 (1968), 138-140.

9. Autoflow. Marketed by Applied Data Research, Princeton, N.J.

10. Steel, T.B. Jr. Standards for computers and information processing. In Advances in Computers, M. Rubinoff and F. Alt, Editors. Academic Press, N.Y. Vol. 8 (1967), 47-152.

11. Brooks, Jack. Letter to Charles L. Schultze, Director, Bureau of the Budget. Comm. ACM 11, 1 (1968), 55-56.

12. Sadowski, W.L., Maximon, L., and Lozier, D.W. A bit comparison program for algorithm testing. Presented at Approximators Workshop, Argonne National Laboratory, April 1971.

13. Abramowitz, M., and Stegun, I. Handbook for Mathematical Functions. AMS 55, National Bureau of Standards.

14. McCarthy, J. A basis for a mathematical theory of computation. In Computer Programming and Formal Systems, North-Holland, Amsterdam (1963), 33-70.

15. McCarthy, J., and Painter, J. Correctness of a compiler for arithmetic expressions. In Proceedings of Symposia in Applied Mathematics, J.T. Schwartz, Ed., Vol. 19 (1966), 33-41.

16. Floyd, R.W. Assigning meanings to programs. In Proceedings of Symposia in Applied Mathematics, J. Schwartz, Ed., Vol. 19 (1966), 19-32.

17. Hoare, C.A.R. An axiomatic basis for computer programming. Comm. ACM 12, 10 (1969), 576-580, 583.

18. Dijkstra, E.W. A constructive approach to the problem of program correctness. BIT 8 (1968), 174-186.

19. Naur, P. Proof of algorithms by general snapshots. BIT 6 (1966), 310-316.

20. Naur, P. Programming by action clusters. BIT 9 (1969), 250-258.

21. London, Ralph L. Bibliography on proving the correctness of computer programs. Machine Intelligence 5 (1970), 569-580.

22. London, Ralph L. Bibliography on proving the correctness of computer programs—addition no. 1. University of Wisconsin, Computer Science Dept., Report No. 104 (Dec. 1970), 1-8.

23. Hull, T.J., Enright, W.H., and Sedgwick, A.E. The correctness of numerical algorithms. SIGPLAN Notices 7, 1 and SIGACT News 14 (1972), 66-73.

24. Ingalls, D.H.H. FETE, a Fortran execution time estimator. Stanford University, Dept. of Computer Science, Report No. 204 (1971), 1-10.

25. Wegstein, J.H. Announcement of algorithms department. Comm. ACM 3 (1960), 73.

26. Knuth, D.E. An empirical study of Fortran programs. Software 1 (1971) 105-133.

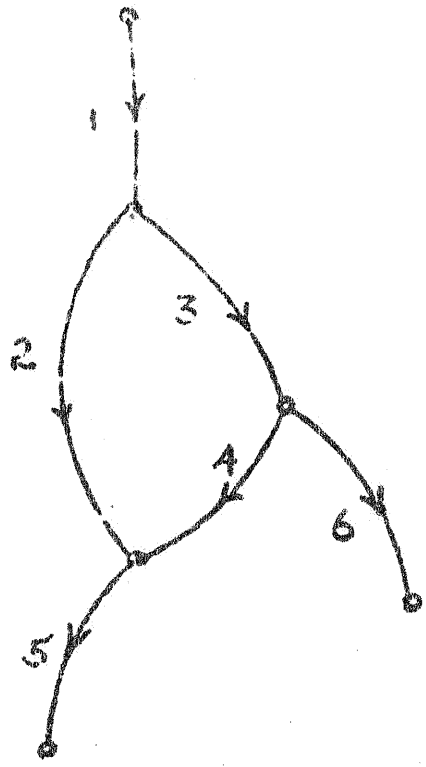
27. Floyd, R.W. Toward interactive design of correct programs. Stanford University, Department of Computer Science, Report No. 235 (1971), 1-12.

28. Wilkes, M.V. The outer and inner syntax of a programming language. Computer J. 11 (1968), 260-263.

29. CALGO, Collected algorithms from CACM. Association for Computing Machinery, N.Y.

30. Rice, J.R. The distribution and sources of mathematical software. In Mathematical Software, J.R. Rice, Ed. Academic Press, N.Y. (1971), 27-41.

FIGURE	CAPTION
1	Program structure with paths: (a) 1,2,5,; (b) 1,3,6; (c) 1,3,4,5.
2	Program structure with paths: (a) 1,5; (b) 1,2,4,5; (c) 1,2,3,4,5.
3	Program structure with paths: (a) 1,3,4; (b) 2,3,5; (c) 1,3,5; (d) 2,3,4.
4	Transition matrix for branch pair traversals: [1,3], [2,3], [3,4], [3,5].
5.	Transformation of the structure in figure made by removal of branch 3.


1
2
3
4
5
6

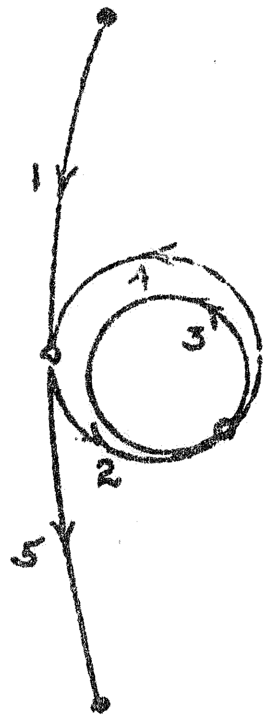


Fig. 2
101

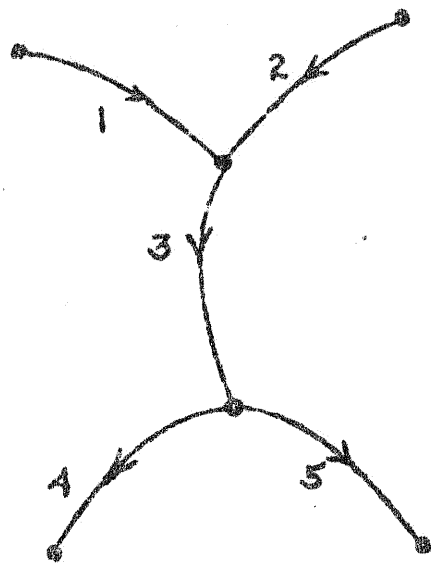


Fig. 3

0	0	1	0	0
0	0	1	0	0
0	0	0	1	1
0	0	0	0	0
0	0	0	0	0

Fig. 4

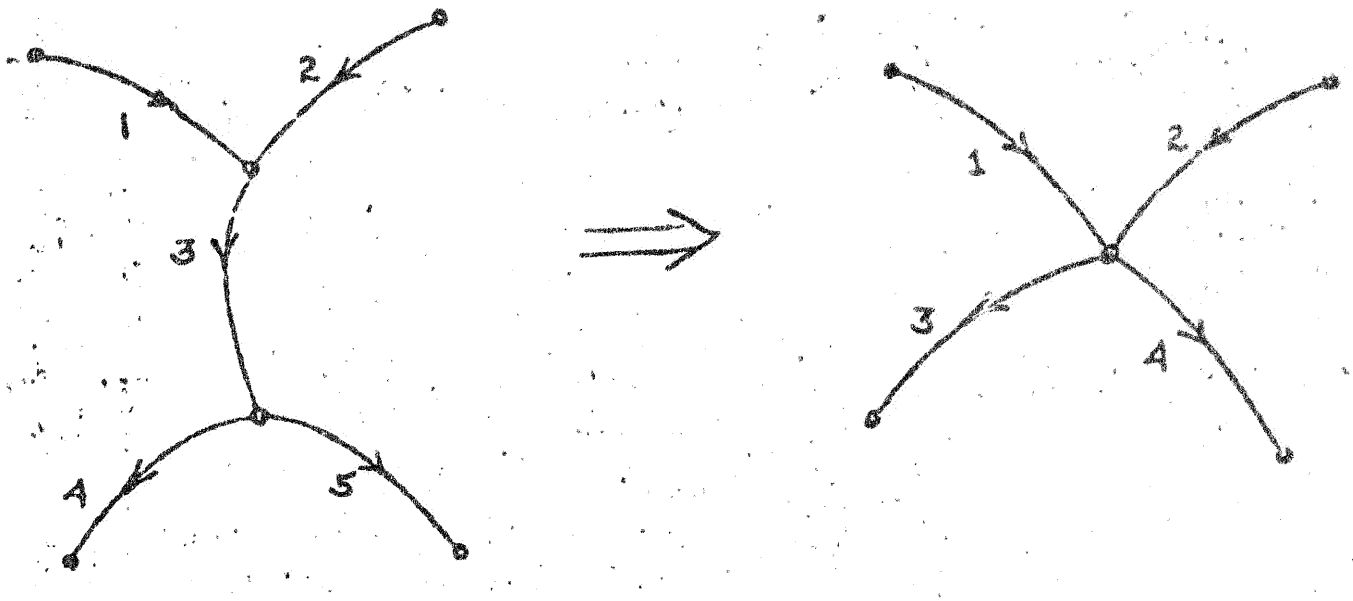


fig. 5
1/5+