

Computer-Drawn Flowcharts*

DONALD E. KNUTH

California Institute of Technology
Pasadena, California

To meet the need for improved documentation of written computer programs, a simple system for effective communication is presented, which has shown great promise. The programmer describes his program in a simple format, and the computer prepares flow charts and other cross-referenced listings from this input. The description can be kept up-to-date easily, and the final output clearly explains the original program. The system has also proved to be a valuable debugging and coding aid.

Introduction

Perhaps the greatest problem in computing today, although little has been written about it, is the need for better documentation of programs. This problem arises in many ways, but basically it boils down to the question: "How can a computer programmer write down the algorithm he has used so that somebody else will readily be able to understand it?"

This problem arises at any computer center where the standard programs have to be documented for future reference. It is especially acute when a computer users group or computer manufacturer distributes programs among installations. It is also important for intercommunication among several programmers working on the same project.

Every group of programmers has of course been faced with this problem and has developed some policy designed to circumvent the difficulties. In most cases, each programmer of the group is expected to follow a set of standard rules for documenting all programs; these rules commonly involve preparation of flow charts. Such a system usually works fairly well (at least as far as the manager of the group is concerned!), but people are beginning to realize more and more that there are shortcomings in the flow chart system:

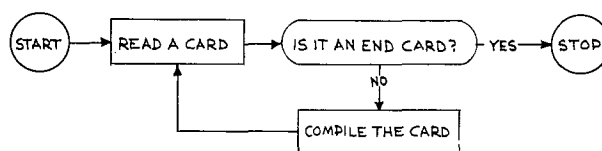
1. *Obsolescence.* Although the flow chart might have described the computer program at one time, a common comment is, "Any resemblance between our flow charts and the present program is purely coincidental." Another frequent remark: "Some day we will update the flow charts." It is expensive to maintain flow charts, yet every change to a program makes the charts obsolete. In fact, busy programmers often retain only the flow chart they used for coding, without incorporating any of the changes which occurred during the debugging stages.

* The preparation of this paper was supported in part by the Burroughs Corporation, and in part by the Evergreen Corporation. The design of the system described was enhanced by discussions with W. C. Lynch and Joseph Speroni.

2. *Lack of readability.* After looking at dozens of sets of flow charts for system programs, I find I have been able to understand only about 25 per cent of them. Apparently brevity is a virtue, and each person tends to make up his own cryptic notation for writing down the information. Elaborate subscripting, superscripting, and Greek letter conventions are created, which are usually quite useless to anyone but the author. This is caused largely by the form of a flow chart itself: there simply isn't room to say very much inside those boxes. Another factor is that flow charts have two purposes: the *creative* flow chart, for helping the programmer get his thoughts in order when initially setting up the algorithm, and the *expository* flow chart, for elucidating the algorithm to someone else. There is no reason that both types of flow charts should be the same; the problem is that the distinction is not clear, and creative flow charts are often passed off as being expository. One frequently hears of computer programs for which "complete flowcharts" are available; fine, you write for and receive copies, but they tell you virtually nothing.

3. *Time consumption.* Programmers have spent many hours with template in hand, drawing beautiful charts on vellum. The fact that this requires a good deal of time tends to provoke a hurried job and a less careful one; thus the obsolescence and lack-of-readability problems are intensified. Even when the charts are drawn by someone else, a great deal of time is required of the programmer, for preparing and proof-reading the copy.

4. *Level of detail.* A wide variation is possible in flow charts. Here, for example, is a flowchart for a compiler,



where a lot of the detail has been suppressed. At the other extreme we find a flow chart with approximately as many boxes as machine-language instructions. To present an efficient exposition, actually several levels of detail are necessary; no one level is sufficient for any but the shortest programs.

Many people have felt that problem-oriented languages, such as ALGOL, COBOL, and FORTRAN, take the place of flow charts. Although programs expressed in this way are somewhat easier to read, it is still a fact that much more information is necessary for someone other than the original programmer to understand the method used. For example, it may take several hours of study to discover how some of the ALGOL algorithms (see Algorithms department of the *Communications of the ACM*) work. This is not a fault of the ALGOL language, of course; it is due to the fact that compiler languages are too detailed a level of description for this purpose.

How can we avoid these problems? A logical approach would be to *let the computer help us*. The computer can at

least handle the more mechanical, clerical details; only the basic ideas should be required of the programmer.

A simple system along these lines was tried on an experimental basis during the summer of 1962. The ideas used were by no means ingenious or completely new; they were merely a combination of several notions which have already appeared in the literature. However, when the system was put into operation, it seemed to "click," and it was extraordinarily successful—much more useful than expected. Therefore we feel it may be the start of something valuable, and it is published here with the hope it will stimulate others to try the system and perhaps to develop it further.

Computers are, of course, widely used today for drawing charts, especially for helping to automate the design of other computers. Circuit diagrams have been prepared by machine for quite a few years [3]. Weather charts, holiday greetings, etc. are produced on the printers attached to computers. An application to program-flowcharts was given by Lois Haibt in 1959 [1]; this is an ambitious program attempting to go from machine language to flow charts automatically, and it is currently in use.

Perhaps the greatest difficulty encountered, if we attempt to have a computer draw flowcharts, is the lack of a large character set. IBM distributes special print wheels designed to help print circuit diagrams (on special order), and perhaps there are other similar devices; but the idea here is to try to do a good job using only equipment which is already available at one's computer center. Although a more extensive character set would be quite helpful, it has not proved to be necessary. A question mark "?" is an especially useful symbol on flow charts, but techniques to avoid using it are not hard to discover. Today's trend is to larger and larger character sets on the new output devices, so things will be improving in this area; the system to be described will, however, work satisfactorily on systems with alphabetic, numeric, and some special characters,

such as a FORTRAN character set. The original system runs on a UNIVAC Solid State computer, whose character set includes no equal sign, but a colon, semicolon, and apostrophe; these more exotic characters were useful but not essential.

This paper begins with a discussion of a three-level system for effective documentation, then describes a simple format for writing algorithms such that a computer can do the rest of the work. Two appendixes appear at the end of the text, for those interested in pursuing the details further: Appendix 1 is a statement of the precise rules of the original flowcharting system, and Appendix 2 is an algorithm by which the reader can set up his own system.

Three Levels of Documentation

Let us try to find a way to present algorithms as effectively as possible. A hint of this appears in a brief article written in 1959, "Flow Outlining—A Substitute for Flow Charting" [2]. The author, W. T. Gant, says the programmers at Shell Oil Corporation found this system "superior to flowcharting, because it is less time-consuming to prepare, easier to code from, and permits more detailed remarks where needed." A *flow outline* is simply a step-by-step, English language description of the algorithm, where every step is numbered or otherwise named.

The difference between a flow outline and a flow chart is essentially that the flow outline is one-dimensional, the flow chart is two-dimensional. For some reason, a two-dimensional, graphical presentation greatly helps to clarify an explanation for human readers. "A picture is worth 1000 words," etc. Therefore, although flow outlines obviously have merit, we cannot expect to do away with flow charts entirely, if we are to have the most effective communication.

An interesting method has appeared in some Russian publications (see, e.g., [5, p. 37]). In this case, the algorithm

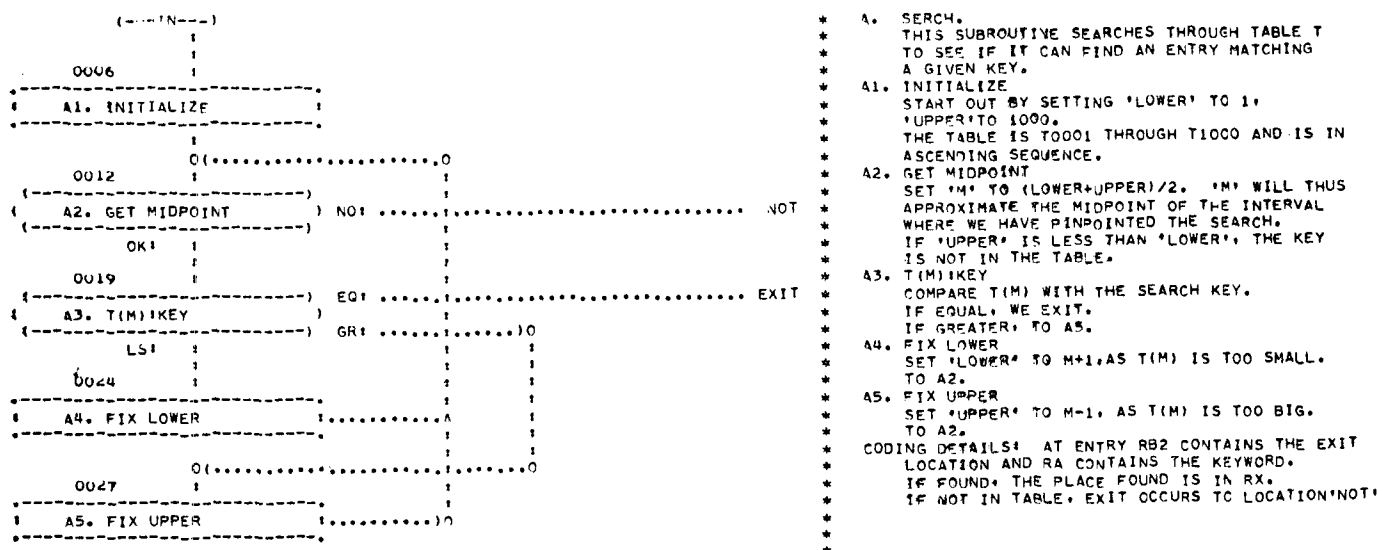


FIG. 1. Flow chart and flow outline for binary search

Experience has shown that a modification of this method is very effective. In this version, the steps in a flow outline are not only numbered, but a short *title* is also given, which summarizes the basic process described in that step. This title or headline should be five words or less (preferably less); its purpose is to indicate briefly what happens at this step in general terms.

Looking at flow charts from this point of view, we see that the graphical, two-dimensional effect is being used to its full advantage; for the effectiveness of charts tends to be inversely proportional to their complexity.

Figure 1 gives an example of such a flow outline, with an accompanying flowchart. The algorithm described is a

Another important point to observe is the type of description appearing in the flow outline. Because the size limitation imposed by boxes is now gone, a clearer explanation of each step is possible. In the flow outline, the programmer should not specify merely *what* is done at that step; it is highly desirable to have some indication of *why* it is being done. Information relating this step to the program as a whole can be given, as well as a description of the current state of affairs and current subgoals at the time this step is reached. One should not merely say, “ J is replaced by $J+1$,” for usually this does not imply much to the reader unless he is keenly aware what J means at this point. Better, perhaps, would be something like this: “We are finished processing the J th item of TABLE, therefore J is increased by 1, in preparation for a new item.”

A great variation in detail is possible here; in general it is preferable to include several related steps in a single flow-outline step. It is even valuable to include some alternative conditions in a single step, e.g. "If N is even then square M , but if N is odd, subtract one from N and double M ." The test whether N is even or odd need not even appear in the flow chart. Such abbreviations are quite often desirable, since a two-dimensional flow is not necessary to clarify such a simple test which can be described in plain terms. On the other hand, there are many applications in which a greater level of detail is

FIG. 2. Assembly language corresponding to Figure 1

desirable for the flowchart, and the programmer is free to choose which he prefers for each case.

A third level of detail is also necessary in a well-documented program, namely the formal, precise language which was input to the computer. In the original system, an assembly language serves as this detailed description, although a compiler language or any other well-defined language would serve as well. Figure 2 shows an assembly language program corresponding to the algorithm in Figure 1. (The computer in this case is the UNIVAC Solid State computer.)

Notice that the same titles and step numbers appear on the assembly language listing as in the flow chart and flow diagram. Furthermore, the numbers just above each box on the flow chart represent the line number of the same step in the assembly language listing. Thus, complete cross-referencing is automatically provided.

It is unnecessary to specify all of the details of a program in the flow outline; only the important ones need appear there. After all, the assembly listing provides the final level of detail, and the flow outline is an informal description. At the beginning of a fairly complicated program, for example, the title in a flow chart box might say, "A1. INITIALIZE." The flow outline might give the additional comment, "Set all pertinent temporary storage locations and counters to zero." The name of all these locations would appear only on the assembly listing.

The example just given should clarify the relationships between the three levels of detail discussed here. The three levels:

- (1) formal language
- (2) flow outline
- (3) flow chart

in increasing order of generality, work together as a team to provide efficient man-to-man communication of algorithms. Experience has confirmed the practical value of this method.

Programmer's Format

The reader may very well ask how all this is going to save him time, if three levels of documentation are now required rather than the one or two now being used. In this section we describe a simple format for writing flow outlines in a way that the computer can readily draw the

10001 LOWER UPPER KEY	FLO BLR 1000 EQU B01A EQU B02A EQU B03A HHH	1999	A.	SEARCH. THIS SUBROUTINE SEARCHES THROUGH TABLE T TO SEE IF IT CAN FIND AN ENTRY MATCHING A GIVEN KEY.
SEARCH	STA KEY LDA#00000 STA LOWER SXL 0300 STA UPPER ADD LOWER	HHH 10000 1F 3F	A1.	INITIALIZE START OUT BY SETTING 'LOWER' TO 1, 'UPPER' TO 1000. THE TABLE IS 10001 THROUGH 11000 AND IS IN ASCENDING SEQUENCE.
1	ATL		A2.	GET MIDPOINT SET 'M' TO (LOWER+UPPER)/2. 'M' WILL THUS APPROXIMATE THE MIDPOINT OF THE INTERVAL WHERE WE HAVE PINPOINTED THE SEARCH.
3	MUL#00000 LDA RA LDL LOWER TGR 2F TEQ 2F	000A5 NOT RA	NO1	IF 'UPPER' IS LESS THAN 'LOWER', THE KEY IS NOT IN THE TABLE.
2	ADD LDA 10000 LDL KEY TEQ 0000 TGR 2F LDA RX ADD#00000 STA LOWER ADD UPPER LDA RX SUB CON 00000	RA 10000 3B 1B 10000	OK1 A3.	T(M) KEY COMPARE T(M) WITH THE SEARCH KEY. EQ1 IF EQUAL, WE EXIT. GR1 IF GREATER, TO A5. LS1
2			A4.	FIX LOWER SET 'LOWER' TO M+1, AS T(M) IS TOO SMALL. TO A2.
			A5.	FIX UPPER SET 'UPPER' TO M-1, AS T(M) IS TOO BIG. TO A2.
NOT TEST -T	HHH HLT LIR1 0000 LIR1 0001 ADD RA STA10000 LIR1 0000 ADD CON 99900 LDA#00010 LIR2 ADD#00000 LIR2AT END TEST FIN	* -T -T 00000 00000 SEARCH 10000 SEARCH	T.	TEST. T1. SET UP T FILL TABLE T, PUTTING 21 IN T(1). T2. SEARCH 100. USE THE SEARCH ROUTINE TO SEE IF 100 IS IN. T3. SEARCH 101. SEARCH ALSO FOR 101 WHICH IS NOT IN THE TABLE

FIG. 3. Input as punched on cards

flow diagrams automatically and can also provide the cross-referencing. The net effect is to save considerable time, while greatly increasing the clarity of the final documentation.

The programmer's first step is to divide the program into logical sections; each section will yield one flow chart. A typical way to make the breakdown is to indicate one section for each subroutine, and one section for each major division of the program. An alphabetic letter is assigned to each section, for reference. (The subroutine in Figure 1, for example, has been designated section A.) The steps in section A are labeled A1, A2, ..., A99.

Each section of the program is an independent unit. If the program is large enough to require more than five sections, a special "preamble section" is given, which explains the basic structure of the program, perhaps gives the format of the files and tables, and shows how data is packed into words. Then a table of contents is given, listing the key-letter and the name of each section. The flow charts and flow outlines of each section follow the introductory information.

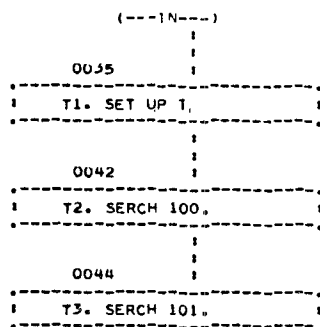


FIG. 4. Another flow chart produced from the input of Figure 3

As in Figure 1, each section usually begins with a description of its general function and some of the assumptions made. At the close of each section another explanatory paragraph regarding the more important coding details often appears.

Figure 3 shows the input as it was punched on cards before feeding it into the present system. Figures 1, 2 and 4 represent part of the output resulting from this input. Only the right-hand side of the input is of concern to us here; the left-hand side is written in assembly language.

There are two fields on the documentation form; the first, consisting of four columns in this case, serves to control the operation, and it is called the "documentation key" field, or DK-field. The remaining field, 45 card columns in this particular implementation, is the "title and remarks" field. In Figure 3, the DK-field can be located as those four columns containing "A.", "A2.", "OK:", etc.

The rules regarding the DK-field are rather simple:

(1) At the beginning of each new section, the letter indicating that section, followed by a period, is put in the DK-field, and the remarks field contains the name of the section. (By the way, this section name will appear on all listings, and it also causes page ejection on all listings so that each new section begins at the top of a page.)

(2) At the beginning of each step, the step number is given in the DK-field, in the form "An." or "Ann.". Then the remarks field contains the title of that step. This title appears on all listings; in this case a title was limited to a maximum of 20 characters, which was always found to be adequate (except for a few cases where 21 characters would have been preferable!).

(3) The DK-field is also used to give names of conditions. Examples of this in Figure 3 are "NO:", "OK:", "EQ:", "GR:", and "LS:". These conditions are transferred directly to the flowchart (see Figure 1), just as they appear in the DK-field.

The final rule for the formatting is the way in which the *successor* of each step is specified. A special character, in this case the number-sign #, is reserved for this purpose and it may not be used in any other way. This special character is punched just preceding the name of the step following the present one. An arrow leading to that step will be drawn on the flow chart. Examples of this in Figure 3 are "#EXIT" and "#A5" in step A3. Notice that the #-sign has been deleted from the actual listing of the flow outline in Figure 1. The name of a successor begins with the first character after the #-sign and continues until the first character which is not a letter or digit, up to a maximum of five characters.

If the only exit from a step is to the step following, no condition is given and no #-sign is used. (This occurs, e.g., in step A1.) If the only exit from a step is to another place, not the step following, no condition is given and the #-sign is used to specify the succeeding step. (This occurs, e.g., in steps A4 and A5.) If there are several successors of a step, condition names are given to distinguish

between branches, and a special shape of box is generated on the flow chart. For each condition name, a successor is specified using the #-sign. If no #-sign appears for a condition, the "next step in sequence" is implied. (For example, consider step A3, where three conditions are given; "EQ:" and "GR:" have out-of-sequence successors specified, while "LS:" refers to the next step. Notice the different treatment given to these condition names in Figure 1.)

If the successor name is of the form "An" or "Ann", where A is the key letter of the *current section*, the successor is somewhere in the same flow chart, and an internal branch line is drawn in the chart. If the successor name has any other form, it is merely placed at the right with a line leading out to it. For example, "NOT" and "EXIT" are such external references in Figure 1, while the references to A2, etc. have been done with internal branch connections.

To summarize this section, we have two main rules:

(1) The DK-field is used for (a) a key letter indicating a new section, (b) a step number, indicating the title line of a new step, or (c) a condition name.

(2) Place a #-sign in front of the name of the successor to a step, unless the successor is simply the next step.

Conclusions

We now report on the use of the original system. As stated before, it saved much more time and gave far better results than were anticipated.

One of the greatest triumphs, perhaps, was that one of the users who frankly disliked documenting programs and usually did so somewhat grudgingly and cryptically, confessed that it was actually fun to document programs by this new method, and he turned out very readable flow charts (for perhaps the first time in his life!).

While trying the system, we used several different approaches. In the original applications, the program was written and checked out first, and then the documentation was written and added to the program. This was in accord with the original intent—merely to save the labor of drawing flow diagrams and keeping them up to date.

But a surprising feature developed. Although the entire plan was oriented towards the preparation of *expository* flow charts, we found that they actually served many purposes of *creative* flow charts. A large number of bugs in the programs were detected during this documentation process, thus saving check-out time on the computer. The following two approaches were found to be most fruitful:

1. First prepare a (sloppy) creative flow chart if you wish, then prepare the code for the program. Before debugging the program, draw the flow outline description for the final documentation, using the handwritten computer code as the source material. Then run this flow outline through the computer, and *debug the flow charts* produced by the flow-charting program. Nearly all of the mistakes in the program are caught in this manner, and it is immediately clear when the flow chart makes no sense.

Since this method is designed for effective man-man communication, it works very well for "man-himself" communication.

2. A second method would be to draw the flow outline and debug it (using the computer-drawn charts) before writing any of the code, then code from the resulting diagrams. Clearly a combination of techniques 1 and 2 can also be used.

Thus we found that our program, which was written purely to help solve the documentation problem, gave us unexpected help in another problem area (namely rapid desk-checking of algorithms) as a free bonus.

Another advantage of the system was that it took little time to prepare. Perhaps it would be of value to give some quantitative time considerations here: We had approximately seven man-months in which to write a FORTRAN II compiler and a complete library of arithmetic and input-output subroutines. These were to be fully documented. (Since we actually worked 12-14 hours per day, seven days a week, the time scale given here is somewhat unrealistic, and perhaps 15 man months would be a truer figure for the total time in terms of an ordinary working schedule; but the actual time spent will be given here.) Absurd as it may seem, we decided to write a complete assembly program as well, and the assembly program was to include an extra pass for drawing these flow diagrams. The times taken for the various stages of the project, including planning, coding and debugging, are approximately:

Card-to-tape and tape-to-tape pass for assembler	2 man-weeks
Basic assembly features	3 man-weeks
Flow-charting portion of the assembler (2 passes)	2 man-weeks
FORTRAN translator	8 man-weeks
FORTRAN loader and storage allocator	4 man-weeks
FORTRAN library subroutines	7 man-weeks
Utility routines for debugging, etc.	2 man-weeks

Each pass of the flow-charting portion required less than 600 lines of coding.

The two man-weeks spent preparing the flow-chart routine were saved many times over; although we cannot be sure, it is likely that we would never have finished if we had not spent nearly one-third of the allotted time preparing auxiliary programs, and the flow-charter in particular. It was very gratifying to see our flow charts pouring out of the printer at 600 lines per minute. All of the programs in the above list are completely documented; the FORTRAN translator has 26 flow charts with accompanying flow outline descriptions. These have been published in limited distribution [4].

Design of the flow-charter changed several times during the course of the summer, until it now has the form indicated in Figures 1 to 4. Since the flow chart program was not our main goal, we did not take the time to dress it up with many frills, or to make major changes in it after it began to work.

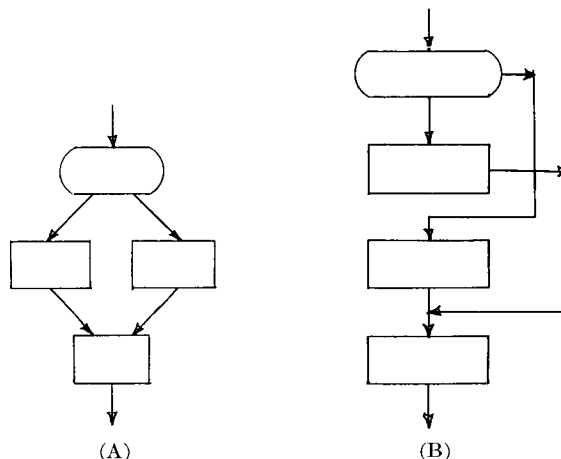
There is one feature in particular that we would now change, based on experience in use. Notice that the condition names, which are written in the DK-field (Fig. 3), are suppressed on the final flow outline (Fig. 1). This occasionally caused peculiar wording to occur in the resulting flow outline, because the programmer forgot to restate the condition in the text. As the system is now (see e.g. step A3), each condition must effectively appear twice, once in abbreviated form in the DK-field, and again in the text. It would have been preferable to reproduce the condition names on the final listing, perhaps separated from the text by an additional blank column.

A further suggestion for future systems is that the flow chart language be divorced from the assembly language, and actually punched on separate decks of cards. Only a DK-field, containing the key letter of a new section and the step number of each step in its proper place, would need to appear on the assembly language cards. The flow chart program would carry out the necessary merging process, or could be used independently for preparing the charts and outlines only, with no formal language description.

The only disadvantage of this dichotomy would be a slightly increased tendency to neglect changing the flow description whenever a change is made in the program. However, the advantages are more significant, as the same basic flow charting program can be made to work with assemblers, compilers and any other formal language system present at an installation.

The flow charts produced by our system are, of course, not as beautiful as those done by a draftsman, but they seem to be quite adequate for their purpose. In order not to be accused of putting good draftsmen out of work, however, we should perhaps add that these diagrams are at least suitable for submission to a draftsman, so that if there is at some point in time very little chance for a flow chart to become obsolete, it can be redrawn in the best professional manner.

A very simple-minded scheme was used for drawing the flow charts: all boxes are in a single column, and all connector lines are found to their right.



this to be quite sufficient, but future systems may wish to add some more topological sophistication. In particular, a fairly common occurrence is something like (A) and in our system the resulting chart is rather clumsily expressed as in (B). A test for special cases of this type might be desirable, as it has been done in [1]. It would not be extremely difficult to incorporate this into the algorithm given in Appendix 2.

It should be noted that even though this system will produce improved documentation, there is still an art to creating an extremely effective presentation. It is possible for a programmer to do a sloppy job with this system if he is so inclined; there is no guarantee of good results. But now that he has a more effective medium for expressing himself, our experience has indicated that he will therefore strive harder to do a better job, thus getting more satisfaction from the result.

We realize, of course, that our system is just a first step, but we feel that it has been taken in the right direction. We also realize that this system will not be *widely* used unless computer manufacturers create and distribute such flow-charting programs to their customers. But in this article it is hoped that a few groups will be tempted into trying the ideas on their own (after all, it is really rather easy to write the program, and a method is given in Appendix 2). Because of our encouraging success, we are sure this will go a long way towards relieving the current documentation headaches.

APPENDIX 1

A few more rules were made regarding the DK-field in our original system; these were not sufficiently important to mention in the main text, and they may be improved upon in future systems.

The DK-field could take the following forms:

1. Blank. No special significance.
2. K. Beginning of a new section, with key letter K.
3. Kn. Beginning of a new step, with this step number.
or
Knn. Step numbers within a section must be in ascending order but not necessarily in sequential order.
4. X Same as blank, except that the entire left-hand part of the card is deleted from the *assembly* listing (as in lines 31, 32 in Figure 2).
5. G The remarks in this card are not part of the flow outline, they are machine-oriented or coding-oriented details which are to appear *only on the assembly listing itself*.
6. TABL Treated as blank; these are the first letters of
or "TABLE OF CONTENTS" and "CODING
CODI DETAILS", respectively, and were allowed in the DK-field primarily for better-looking output.
7. Other A condition name. Although all the condition names in Figure 1 have colons, this is by no means a requirement. It would have been nicer to have allowed five columns for the DK-field, to allow longer condition names.

It is possible to assemble with or without drawing flow charts. When flowcharting is not in operation, the DK-

field and remarks are simply treated as a standard comments area on an ordinary assembly program listing. When flow charting, most of the comments are suppressed from the assembly listing (see Figure 2). This makes the assembly listing more readable, but it also makes it much harder to make corrections to the decks when changes are necessary. Therefore we found it wise to assemble both with and without flow charting, marking our corrections on that listing which included all the punches on the cards. This is another reason why it would be wise to separate the documentation information from the assembly information: corrections will be more simply made.

NOTE. The assembly program described here is for a rather unusual machine configuration (Solid State II-80, with 8800-word drum, 1280-word core, and 6 tape units), and it was designed merely to help create the FORTRAN II compiler rather than for distribution of the assembler itself. Therefore the assembly and flow-charting system are not a part of the UNIVAC Solid-State Library.

APPENDIX 2

If this method for explaining algorithms has any merit at all, we should at least use it in this article to explain the flow-charting algorithm itself. Due to lack of space, however, a somewhat abbreviated description of the algorithm must appear here. We describe a method which is independent of an assembly program, as suggested in the last portion of the text; the only connection with other programs is that we assume a reference number may be available for each step, indicating a line number on some other listing.

The algorithm proceeds in two passes. The first pass digests the information and edits it into a convenient form, then the second pass produces the actual listing. In the present system, the listing contains the flow chart at the left, and the flow outline reproduced at the right. For simplicity, we will only describe a program to print the flow charts; the rest of the program, which simply copies the input but suppresses the *%*-signs, is a straightforward addition (the only complication being to properly terminate a chart when both halves of the listing are finished). A simpler alternative would be to print the flow outline listing during pass 1, and to print only the flow charts in pass 2. This was not done in our system since we printed the assembly listing during the first pass.

Some intermediate language must be devised for any two-pass algorithm, in order to communicate the information from one pass to the other. In this case, as in many others, it is convenient to make this an *interpretive* type of language. The first pass "compiles" into this interpretive pseudocode, and the second pass is merely an *interpretive routine*, executing the pseudocode instructions.

The instructions in this pseudocode have the general form (OP, ADDRESS), although other information is also intermixed with operators, as will be evident in the discussion which follows. The details of the operators are as

follows:

(1,n) Prepare a square box for step Kn. (K denotes the key letter of the current section.) The next line of code is the line reference number corresponding to the formal language listing. The following five lines contain 25 alphabetic characters to insert in the flow chart box.

(2,n) This instruction is exactly the same as (1,n), except a branch-shaped box is produced rather than a square box.

(3,0) Terminate this flow chart, and get ready to begin another.

(4,0) Terminate this flow chart, and then stop everything.

(5,n), (6,n), (7,n) Draw a branch to step Kn. The next line contains a five-character condition name to identify the branch. OP 7 is used for the *first* branch if there are more than one; OP 5 is used for the *last* branch, if there are more than one, and OP 6 is used for any other branches. If $n = 0$, an extra line of code appears, giving the *name* of the place branched to (the branch is to a step external to the present section).

(8,0) Label the branch to the next box; the following line has the condition name to be used as a label.

(9,n) Draw an unconditional branch to step n. If $n = 0$, the next line has the appropriate successor name.

This code can be explained most clearly by exhibiting the pseudocode corresponding to Figure 1:

Location	Instruction	Location	Instruction
01:	(1,1)	26:	bbbbbb
02:	0006	27:	(7,0)
03:	bA1.b	28:	bEQ:b
04:	INITI	29:	bEXIT
05:	ALIZE	30:	(5,5)
06:	bbbbbb	31:	bGR:b
07:	bbbbbb	32:	(8,0)
08:	(2,2)	33:	bLS:b
09:	0012	34:	(1,4)
10:	bA2.b	35:	0024
11:	GETbM	36:	bA4.b
12:	IDPOI	37:	FIXbL
13:	NTbbb	38:	OWERb
14:	bbbbbb	39:	bbbbbb
15:	(6,0)	40:	bbbbbb
16:	bNO:b	41:	(9,2)
17:	bbNOT	42:	(1,5)
18:	(8,0)	43:	0027
19:	bOK:b	44:	bA5.b
20:	(2,3)	45:	FIXbU
21:	0019	46:	PPERb
22:	bA3.b	47:	bbbbbb
23:	T(M):	48:	bbbbbb
24:	KEYbb	49:	(9,2)
25:	bbbbbb	50:	(3,0)

In addition to this pseudocode, a table LREF with 99 entries is transmitted to the second pass, where LREF(n) is zero if no branch lines occur to step Kn, otherwise LREF(n) is the location of the instruction on which the vertical line connecting to box Kn is to be discontinued. In this case we have LREF(n) = 0 except LREF(2) = 49, LREF(5) = 42.

Armed with this information, the second pass will not need to look ahead, and it can print the information at high speed.

Pass 1. The algorithm for the first pass could be written as shown in Figure 5.

A1. INPUT NEXT CARD

Read in the next card image. (If there are no more cards left,

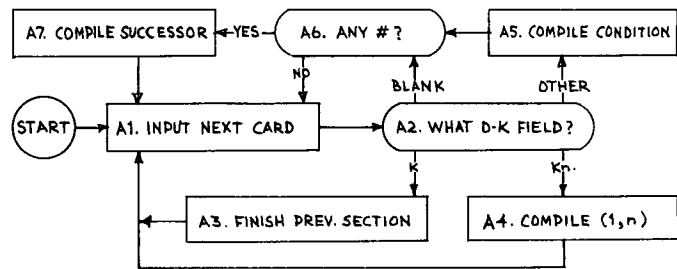


FIG. 5. Flow diagram for first pass of algorithm in Appendix 2

compile a (4,0) instruction, then dump out the LREF table and all the pseudocode for the previous section, and transfer to the second pass of the program.)

A2. WHAT DK-FIELD?

If the DK-field is of the form "K." go to step A3; if it has the form "Kn." or "Knn.", go to step A4; if it is blank, go to step A6; otherwise the DK-field is assumed to contain a condition name, and we go to step A5.

A3. FINISH PREV SECTION

Compile a (3,0) instruction, and dump out the LREF table plus all the pseudocode for the previous section onto tape; this will be sent to the second pass. (This step is bypassed the very first time, since there was no previous section.) Then record the new key letter, set the entire LREF table to zero, and get ready to begin a new section. Go back to step A1.

A4. COMPILE (1,n)

We are at the beginning of a new step. If the preceding step had a condition name branching to here (i.e., not followed by any successor indication), compile (8,0) followed by the condition name. In any event, compile a (1,n) operation, followed by the line reference number and the five words of alphabetical information in the title of this step. If LREF(n) is not zero, set LREF(n) equal to the location of this (1,n) operation code. Return to step A1.

A5. COMPILE CONDITION

If a condition preceded and had no named successor, save its condition name (which will be used later to form an (8,0) operation the next time step A4 occurs). Increase the last operation code by 1. This will change 1 to 2, 5 to 6, or 6 to 7. If the last operation had been a 1, compile (6,0), else compile (5,0), followed by the condition name. These manipulations will cause the following sequence of operation codes in the pseudocode:

If there are no conditions: 1;
 If there is one condition: 2,6;
 If there are two conditions: 2,7,5;
 If there are three conditions: 2,7,6,5;
 If there are four conditions: 2,7,6,6,5; etc.

The different numbering of branch operation codes 5,6,7 is used to control where the condition name is placed on the charts by the second pass.

A6. ANY #?

Search the remarks field to see if the character "#" occurs. If not, return to step A1.

A7. COMPILE SUCCESSOR

If the preceding operation code is 1, this is an unconditional branch, and so a (9,0) instruction is now compiled. Determine the successor name (the characters following the # sign). If it has the form Kn or Knn, where K is the current key letter, change the address of the previous pseudocode instruction to n, and set LREF(n) equal to the location of that instruction. Otherwise,

compile the successor name into the pseudocode. Return to step A1.

Pass 2. As mentioned before, pass 2 is a type of interpretive system. For simplicity, assume that the character set is that used in Figure 1.

The only complexity in this pass consists of controlling the lines to be drawn. This is handled by considering 15 sets of 2-column pairs; each pair is set to "ON" or "OFF" or "SPECIAL." The operation of "ON" and "OFF" varies depending on whether a horizontal line is crossing this column or not:

	OFF	ON
horizontal line	..	::
no horizontal line	bb	b:

The meaning of "SPECIAL" is that one of the following 2-character pairs is used

)0 .0 .V .A

and that a horizontal line is suppressed to the right of this column. Furthermore, this column is to be set to OFF or ON for the following line.

Besides these 15 columns, there is a vertical line for connecting a box to that below it; this is also said to be either OFF or ON.

A subroutine called ASSIGN is used to handle horizontal lines. If an external reference is to be made (e.g. to "EXIT"), a horizontal line is simply run all across the page. Otherwise an internal reference is being made, e.g. to Kn. First the subroutine checks whether one of the 15 columns is already in use for Kn. If not, a new column is chosen (the first available column of the set 1,5,9,13,3,7,11,15,8,14,2,12,4,10,6 in that order of preference) and it is given the SPECIAL status "0". If a column has already been assigned, however, check LREF(n) to see if this is the place where the vertical line is to be stopped. If so, the SPECIAL status "0" is given to this column; otherwise either ".V" or ".A" is given, depending upon whether flow is currently going down or up this line.

The ASSIGN subroutine is used when a condition name is to be processed: a column is ASSIGNED, and a horizontal line is run up to this column. The ASSIGN subroutine is also used when bringing a horizontal line *into* the flow, except here the special status "0" overrides the status chosen by the algorithm in the preceding paragraph.

The subject of the last two paragraphs is hard to explain briefly, but the example of Figure 1 should help in clarifying the situation. At the beginning, all 15 columns are in the OFF status. Then at line 9 of that chart, the ASSIGN subroutine is first used to choose a column for step A2. Column 1 is chosen, and it is given the special status "0" which is later overridden to be ".0" since it is an input line. After line 9, column 1 remains in the ON status. In line 19 the ASSIGN subroutine is used to select a column for step A5. Column 5 is chosen, and given the status "0", remaining ON afterwards. In line 24 the ASSIGN subroutine is used for step A2; column is already assigned for A2, and therefore the status ".A" is given to this column here (the flow is going upwards). In lines 27 and 30, the ASSIGN subroutine is used again, for steps A5 and A2 respectively, and in both cases the LREF table indicates that the column is to be OFF after that reference.

A line-by-line description of the procedure followed for the operators (1,n) or (2,n) follows:

First line (input node): If LREF(n) = 0 there are no branch lines leading into this node, and no special action occurs. If LREF(n) ≠ 0, the box-connector line is set to ON, and the ASSIGN subroutine is used as described above to run a horizontal line to it.

Second line (cross reference number): On this line the cross-reference number is obtained from the pseudocode and put onto the listing.

Third line (top line of box): The top line of a box is created. If this operation is (1,n), turn ON the box-connector line, and draw a square box; if this is the operation (2,n), turn OFF the box-connector line, and draw a rounded box. If the next opera-

tion in the pseudocode is of the form (7,n), this condition name is also processed for this line.

Fourth line (alphabetic): The alphabetic title is put into the box. If the next operation in the pseudocode is of the form (6,n), this condition name is also processed for this line. If the next operation is of the form (9,n) an unconditional branch is made (as if the condition name were "...") and the box connector line is turned OFF.

Fifth line (bottom line of box): The bottom line of a box is created. If the next operation is of the form (5,n), this condition name is processed. If the next operation is of the form (6,n) there were more than four conditions; extra lines are added, one condition per line, until a condition of the form (5,n) is finally processed.

Sixth line (possible label): If the next operator is of the form (8,0) the label is inserted now and the box connector line is turned ON.

Transfer to next operation: If the bottom of the page is dangerously near, print lines which are blank (except for the vertical lines which are currently ON) until the top of the next page is reached. Then examine the next line of the pseudocode: the operator must be less than or equal to 4, or else an error has occurred. Process the next operator.

Comparison with Figure 1 will illustrate this procedure. The essential feature of the algorithm is that it processes each line by itself and then prints the line, rather than consuming memory space to store a whole flow chart before printing it out.

REFERENCES

- HAIBT, LOIS M. A program to draw multilevel flow charts. Proc. Western Joint Comp. Conf., 1959, 131-137.
- GANT, W. T. Flow-outlining—a substitute for flow charting. *Comm. ACM* 2 (Nov. 1959), 17.
- AARONSON, DAVID A., AND KINNAMAN, CLARISSA J. Production of large and variable size logic block diagrams on a high speed digital computer. AIEE paper CP 61-1116, Oct. 1961. (This paper includes a valuable bibliography of the subject.)
- FORTRAN II routine block chart (annotated). Document UP-3843.1, Univac Div. Sperry Rand Corp., 1963. (To adequately understand this document, the reader should be familiar with the FORTRAN II input language of this particular implementation, and with the UNIVAC Solid-State computer machine language.)
- ERSHOV, A. P. Programming Programme for the BESM Computer. Tr. from Russian by M. Nadler, Pergamon Press, 1959.
- Towards better documentation of programming languages, a series of eight papers, *Comm. ACM* 6 (Mar. 1963), 76-92. These articles are primarily concerned with the documentation of *languages* and computer systems for the user, rather than with the documentation of the *techniques* used in the programs themselves as described here.
- IBM 7090/7094 IBSYS-IOEX Programming System Analysis Guide. IBM Form C28-6299, IBM Corp., 1963. This document, which has just come to the attention of the author, includes flowcharts printed in a pleasing format by computer. Unfortunately no mention is made of the input language for this program.