N73-19224

A SCAN PROCESSOR AS AN AID TO PROGRAM DOCUMENTATION

Dr. Paul Oliver
UNIVAC

Documentation is an integral part of program development. It serves as a link between the programmer and the program user or analyst. Good documentation provides the information necessary to use or analyze the program. The investment of program development costs is protected by a document that meets the needs of the potential program user. The following reasons (ref. 1) are generally given to justify documentation:

(1) Documentation is a permanent record that is used in debugging, as a source of future reference, to reduce cost of personnel turnover, and as a project history.
(2) Documentation encourages standardization of coding conventions and the description of computer operations.
(3) Documentation provides the means with which to estimate the extensiveness of program changes and to schedule computer operations.
(4) Finally, and perhaps most importantly, documentation represents a communication link with other programmers and with the nonprogramming community.

The case for documentation is a valid and substantial one but cannot be universally applied. The cost of documentation is certain, but its use is not. It can be safely said that heavily used programs implemented on large configurations should always be documented, as should interactive systems because the nonlinear nature of such systems makes them unusually difficult to analyze and debug without adequate documentation and because of the cyclical nature of production work usually found in business or administrative data processing installations.

Program documentation can be separated into two categories: documentation for program use and documentation for program analysis, modification, or extension. The former contains detailed instructions to evaluate the program's capability and to use the program readily, whereas the latter should provide a detailed development of the problem and program logic. This paper concerns itself only with documentation aids for program analysis.

Documentation can be broken down into chronological phases. The documentation to be performed during the program design and planning stage is probably the most important but is not readily amenable to automation. Postmortem documentation is also important, but aids in this area involve mostly text-processing systems, which are outside the scope of this conference. The programmer can best be helped in the documentation process during the programming.

Despite its importance, it is a well-known fact that documentation is woefully neglected. Even the well-intentioned programmer can always find more urgent demands for his time. It is, therefore, important that a system be provided whereby his program deck, e.g., FORTRAN source code, can be operated on by a processor whose output would provide meaningful documentation, much as the deck is operated on by the compiler. This would remove much of the documentation burden from the programmer. Use of the processor would, of course, require that certain conventions and practices be followed in the coding, but these can be kept at a minimum. An example of such a processor would be one that produces a flowchart of the given program. Several such processors are available, although most are of dubious quality. The production of a flow diagram is certainly one of the functions that the processor should perform. A more fruitful function would perhaps be the scanning of a collection of source language statements to produce listings of the variety of symbols found, arranged in any of several ways that might suit a user's purpose and related to the lines of coding in which the symbols occur. Thus far features to produce indication of general program flow (the flowchart production subsystem) and a tabular analysis aid (the symbol scanner) have been included. A concise representation of the "decision stations" in a given program should also be included in the documentation. Decision tables provide an attractive way of accomplishing this and can be produced in an automatic fashion relatively easily (ref. 2). A decision table subsystem is also included as part of the scan processor. Each of these functions and subsystems shall now be examined in turn.

THE FLOWCHART PRODUCTION SUBSYSTEM

A flowchart is one of the means available by which visual representations (the block diagram) of relatively abstract concepts (the programs or systems) can be provided to programmers, analysts, and managers. Flowcharting has been documented ad nauseam, and such documentation will not be repeated here. The reader unfamiliar with the American National Standards Institute (ANSI) standard flowchart symbols and their usage will find reference 3 useful. It suffices for the purpose of this discussion to say that flowcharts show the path of data as they are processed by a system or program, the operations performed on the data, and the sequence in which these operations are performed. One generally distinguishes between a system flowchart which describes the flow of data through an entire system, and program flowcharts, which describe what takes place in a program. Program flowcharts are the only concern of this paper.

There are several flowcharting programs available from computer manufacturers or independent software firms. The quality of the flowcharts produced varies considerably among these various sources, and there appears to be little in the way of standardization. This is not overly disturbing because some do not believe that flowcharting needs to be standardized. This attitude is generally taken by those who regard flowcharting as a very "personal" thing. Once the automatic production of flowcharts is discussed, however, this is no longer a personal matter. Thus, one of the features of the flowcharting subsystem is that the ANSI convention should be followed, although the system need not be capable of producing all the standard ANSI symbols. The majority of flowcharting needs could be

satisfied by the basic outlines for input/output, flow, and processing. To these would be added the outlines for connectors, decisions, subroutines, and terminal points. It would also be desirable to include some (currently) nonstandard symbols to reflect characteristics of higher level languages, such as vertical parentheses (block structure such as that present in ALGOL or PL/I) and DO loops. Historical circumstances have resulted in flowchart symbols that are often more suited to describing assembler language programs than FORTRAN programs, for example.

Other desirable products of the flowchart subsystem would be the option of producing a source listing of the program being processed. It would also be desirable to obtain listings of all jumps, for example, as results of GO TO and IF statements, sorted by source and destination of the jump, and of all statement labels or numbers encountered. Box numbers should be included in the printed flowchart, and these numbers would be included in the above listings. Thus, in the listing of all labels and statement numbers, there would also be an indication of the flowchart box number pertaining to a given label or statement number.

These features are by no means exhaustive of those possible in a flowchart program or, in fact, of those available in existing programs, but they are sufficient to produce a flowchart that provides meaningful information about the program. Furthermore, application of these features would require no more than the invocation of the flowchart subsystem on the part of the programmer. Such features as options to indicate the type of box to generate for a given statement (overriding the standard option) or options to control the analysis of instructions could also be included. These may indeed be useful, but they would require the programmer to specify these options in his program, which would alter the program itself and thereby defeat the very aim of an automated documentation process. If the programmer were willing to take the time to specify options and provide details to the processor, the processor would not be needed in the first place because it would be as easy for the programmer to take that very same time and produce the flowcharts with pencil and template.

THE SYMBOL SCANNER

Broadly speaking, the purpose of the symbol scanner is to scan a collection of source language statements and produce a sorted listing of the symbols found, the programs or subroutines in which they were found, the lines in which the symbols are defined, if applicable, and the lines in which the symbols are referenced. The scanner must be a general-purpose one in the sense that it should be usable on a variety of higher level language programs as well as on a given assembler language program.

It is important that the user be given the option of specifying which symbols or classes of symbols are to be included or ignored during the scan. This is particularly important for debugging purposes. The user could, for example, identify all program loops by making one pass on the program during which only the symbol DO is looked for. Likewise, he could identify all possible sources of a floating-point comparison error by performing a scan for symbols beginning with numerics only. The default option would be to include all symbols in the scan. A first-level selection capability could be provided through options

that specify "ignore strings beginning with an alphabetic character" or "ignore strings beginning with numeric or special characters." Finally, a more detailed selection capability could be provided enabling the programmer to specify, through data cards, that specific symbols are to be ignored or that only certain symbols are to be included in the scan.

The listing produced by the symbol scanner would be useful in program optimization as well as debugging. The placing of statements such as "PI = 3.14159 . . ." in a FORTRAN DO loop is a well-known faux pas, but one which nevertheless often occurs. Many compilers will catch such misdeeds, but some will not. A listing of all occurrences of loops in a FORTRAN program may encourage the programmer to perform a little nonautomated optimization of his own.

DECISION TABLE SUBSYSTEM

Decision tables have been known and used for some time by programmers and systems analysts involved in business or administrative data processing. However, their use is not widespread among programmers in general. This is regrettable because decision tables constitute an excellent way of assembling and presenting related items of information to express complex decision logic in a way that is easy to visualize and understand. Complex programs, such as those associated with interactive display systems, are rendered complex by the torturous decision logic present. Decision tables are a powerful tool with which the programmer or analyst can follow the labyrinths of complex programs.

In addition to illuminating decision logic, decision tables have the distinct advantage of being understandable to a nontechnician like a manager or administrator.

Essentially, decision tables can indicate "if . . . then" relationships occurring in a program. The structure and use of decision tables are adequately described in the literature (ref. 4). The following example should suffice to give the unfamiliar reader a feel for the decision table format. Consider these lines of FORTRAN coding:

```
      IF (A.EQ.B) GO TO 15
      X =  5
      Y = 10
      GO TO 20
15    IF (C.LT.D) GO TO 25
      X = 10
      Y =  5
20    RETURN
25    X = Y
      RETURN
```

The decision logic of this short piece of programming expressed in decision table format is shown in table 1.

Table 1.—Decision Logic

Condition	Rule number			
	1	2	3	4
A.E.Q.B.	Y	Y	N	N
C.LT.D	Y	N	Y	N
X = 5			X	X
X = 10		X		
X = Y	X			
Y = 10			X	X
Y = 5		X		
GO TO 15	X	X		
GO TO 20			X	X
GO TO 25	X			
RETURN	X	X	X	X

The horizontal and vertical double rules serve as demarcation: Conditions are shown above the horizontal double rule; actions, below; the portion to the left of the vertical double rule is called the stub, and the portion to the right consists of entries. Each vertical combination of conditions and actions is called a decision rule.

Table 1 is of the limited entry type. The entire condition or action is written in the stub, and the entry shows only, for each case, whether the condition is true, false, or not pertinent (Y, N, or blank) and whether a particular action should be performed (X or blank). An extended entry table would show part of a condition or action in the entry side of the table. Also, numbers indicating the order of a set of actions could be used in place of the X's. A mixed entry table is a combination of these two types.

It should be clear from this brief example that a limited entry decision table would be quite easy to construct and present as printer output. The information necessary to construct the table is easily obtainable, and the format lends itself to printing on a standard printer. It would be desirable to produce such tables in a modular fashion. A single table might be produced for a given program showing only decisions causing transfer of control. Then, a decision table for each of the transfers would be produced giving the detailed decision logic for the corresponding segments of coding.

It might be argued that decision tables would be redundant in light of the inclusion of the flowchart subsystem. Such is not the case. The flowchart's primary purpose is to provide a visual representation of program flow, of which decision points are only a portion. In contrast, decision tables isolate the decision points, giving only the decision logic of a program, unencumbered by other particulars. Rather than being redundant, these two forms of program representation are complementary.

INVOKING THE SCAN PROCESSOR

This special, documentation-producing system could be used in much the same way as one calls a compiler. This could possibly, perhaps probably, result in its seldom being used because a distinct effort would be required on the part of the user. Perhaps a more fruitful

approach would make the calling of the scan processor an option on the compiler or assembler request card. The scan processor could then be implemented as a subsystem of the compiler or assembler for a given language. In addition to making the processor easy to use, this approach would take advantage of the fact that the information required by the three subsystems discussed above is generally obtained as part of the compiling or assembling procedure. The additional overhead incurred when the documentation option is exercised as part of a FORTRAN compilation, for example, could be decreased by such means as allowing the user to specify the number of columns per card to be scanned; for example, 72 for FORTRAN.

SUMMARY

The proposed processor would provide simple yet meaningful documentation for program analysis in the form of flowcharts, a "dictionary" of symbols, and decision tables. This documentation would be obtained with a minimum amount of effort on the part of the programmer and would be called from the program source deck itself. This is an important point. Each of the proposed subsystems could be far more sophisticated and comprehensive than is suggested here. This would in turn require a considerable increase in effort on the part of the user, and experience has shown that the amount of documentation attempted by a programmer varies inversely with the amount of effort required. It is also important to note one glaring shortcoming of the system proposed. The scan processor would give little or no information on data representation. This is a serious omission because data representation is the very essence of programming. Unfortunately, the documentation of data allocation and encoding does not readily lend itself to automation and will have to depend on the doubtful diligence of the programmer.

The processor suggested here would provide minimal documentation for use by programmers, analysts, and management. It would also, hopefully, provide an aid to the manual production of comprehensive, professional, and standardized program documentation.

REFERENCES

1. Chapin, Ned: Paper presented at ACM Professional Development Seminar on Documentation Techniques (Washington, D.C.), 1969.
2. McDaniel, Herman: *Decision Table Software.* Brandon Systems Press, 1970.
3. Chapin, Ned: Flowcharting With the ANSI Standard: A Tutorial. ACM Computing Surveys, June 1970.
4. McDaniel, Herman: *An Introduction to Decision Logic Tables.* John Wiley & Sons, Inc., 1968.

DISCUSSION

MEMBER OF THE AUDIENCE: I would like to comment upon the presentation. I appreciate very much your introduction of decision tables in this. I think that decision tables are really probably the best way to document a program, show the analysis, and essentially wrap up a lot of this stuff very simply. It solves a lot of the problems that occur with flowcharts. The length of data names is no problem, and they can be as long and as descriptive as desired. You have programs that process these decision tables directly in the code. They are very easily checked again for errors and logical omissions, etc. I would really like to see

someone take these decision tables, which are essentially self-documenting, and possibly go back from the source code to decision tables.

DR. OLIVER: I think, quite frankly, that the business community knows a great deal more about data processing than the R&D community. Documentation to them is a money matter, a practical matter, a managerial matter. So business does not object to simple and economical solutions to documentation.