X-543-73-62
PREPRINT

AUTOMATED METHODS OF COMPUTER PROGRAM DOCUMENATATION

(NASA-TM-X-66196) AUTOMATED METHODS OF COMPUTER PROGRAM DOCUMENTATION (NASA) 245 p CSCL 09B

G3/08

N73-19204 THRU N73-19224
Unclas
64254

NOVEMBER 1970

GSFC — GODDARD SPACE FLIGHT CENTER
GREENBELT, MARYLAND

X-543-73-62

AUTOMATED METHODS OF COMPUTER PROGRAM DOCUMENTATION

November 1970

The proceedings of a symposium held at the NASA Goddard Space Flight Center, November 3 and 4, 1970

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

FOREWORD

With the increasing complexities of computers and software, the annual cost of computer software is rising more rapidly than that of hardware. Means must be found to reduce this cost. One fruitful area appears to be documentation. Well-documented, reusable programs should reduce the cost of programming because new work can often be built around existing modules. However, documentation is generally postponed until the work is completed. Then, because of the pressure of new work, the documentation is often neglected or not completed.

If means could be devised for automating portions of program documentation, the time and effort needed to complete documentation would be minimal. This could then lead to greater standardization of listings, tables, segregated routines, input/output format descriptions, and setup sequences, and other advantages. It could then be expected that programmers using the automated system could more easily provide acceptable documentation and make the results of their work more available to others.

This symposium was designed to cover as broad an area as possible on methods for automated documentation of computer programs. There is much in common between different methods of documentation, and among them considerable cross-fertilization is possible.

These papers were presented at the Goddard Space Flight Center, Greenbelt, Maryland, November 3 and 4, 1970. It is hoped that this compilation of current ideas will encourage further progress in automatic documentation efforts.

Preceding page blank

ACKNOWLEDGMENTS

Darrin H. Gridley, Associate Chief, Computation Division, and Evmenios P. Damon, Head, Computer Systems Branch, Computation Division, both from Goddard Space Flight Center, had prime responsibility for organizing this symposium. They wish to thank the many participants for their knowledgeable viewpoints on the new approach to a very difficult subject.

CONTENTS

Session I

PROGRAM DOCUMENTATION WITH ADVANCED DATA MANAGEMENT SYSTEMS

Dr. Wayne B. Swift
Computer Sciences Corp.

The problems of program documentation posed by modern data management systems (DMS) are becoming increasingly important as the use of such systems becomes more prevalent. There seem to be two types of program documentation: the kind that should be done and the kind that usually is.

This first kind of documentation occurs when programmers begin a development by preparing requirements documentation as the very first step. They first develop the top level specifications that describe what the program will do. Once it is agreed that this is, in fact, what the program should do, the next level of system design, requirements analysis followed by general design, begins. After these steps have been approved, the design is broken into smaller and smaller pieces. The process of breaking down the design into small pieces is analogous to the engineering practice of detailed design. The process continues until the pieces are small enough for someone to prepare. A document that sets forth the segmented design is thus automatically created before any code is actually written.

What usually happens is that a programmer decides what a program is supposed to do, writes the program, checks it, and satisfies himself that the program does what is required. The documentation that results from this approach usually consists of only scattered notes plus a few general reports that do not fill in all the gaps in the development. At this point, it is usually quite difficult to improve the documentation because the programmer is probably heavily engaged in a new project or has left to join another organization.

The computer business has liked to think that this kind of documentation has fallen into comparative disuse recently. It is still, however, far from dead and probably the most prevalent kind, though everyone in the computer business reports that plans are under way to change over to good practices very soon. Manual documentation is generally so bad that even very poor automatic documentation is often better, which underscores the need to press for almost any kind of automatic supplement to or substitute for manual documentation. If any help at all can be given to the automation of requirements documentation, it may also serve to encourage better documentation practices.

The automated documentation aids that are generally available today, however, are primarily useful at or near the coding level. Automatic flowcharting and the like are, of course, quite useful in their own way. Full use of all such aids should be pushed, but this will not help much in settling requirements and assisting in the development of direct documentation practices of the type that should be followed. This is true even in the traditional

Preceding page blank

areas of program development where a programmer works on a single job at a time, at least from a logical point of view. This symposium will deal, to a great extent, with available tools in circumstances that include a computer with a normal operating system and the usual associated software. This paper, however, will consider those cases in which the programmer is using a computer along with an advanced DMS.

This situation presents two types of severe special documentation problems. One has to do with the high-level description of the approach that the programmer takes in processing data with the aid of the DMS. The other problem is attempting to figure out what the program has actually been doing. During debugging, many tools are required; this also affects the general documentation problem. Tools that adequately reveal what has been done are, in general, not very widely available with the present generation of DMS's.

There is, perhaps, a third type of problem related to both of these: Given a large and generalized DMS, how does one prove conclusively that this system either does or does not do exactly what it is supposed to do. Certainly, the answer to this problem involves documentation.

A DMS is used to produce a wide separation between procedures, on one hand, and data, on the other. This separation itself creates a considerable number of documentation problems that are not found in normal computer processing. In proposing DMS-oriented, automatic documentation tools to solve some of these problems, five specific problem areas for which some better kind of tool ought to be found come to mind.

The first of these areas is data description. A DMS facilitates data description by separating the problem. Data description lends itself to generalized documentation much better than does process description. It seems quite possible that a generalized data description language may be developed that can itself provide adequate documentation of data without any automated documentation at all beyond the *use* of this language itself. In spite of the fact that processing languages (for characterizing things that are done to data) seem to be evolving toward a babel of different languages tailored for different purposes, there is real promise for the development of a single generalized data description. This will be useful at even the highest level of requirements specification.

The second problem area is the process description language at high macrolevel, really the level of general design or perhaps even requirements specification. Such languages are likely to be intended for application areas only. At this highest level of characterizing what a system does, tools of representation can be automated either by the development of synoptic representations of the logic flow or by the development of languages that are more readily understandable than are typical programming languages. The potential for this is great, but its realization is much further away than is the development of data description languages.

The other three problem areas to be discussed all fall into the general area of execution-time documentation, as opposed to the overall descriptive documentation. Increased use of automatic documentation tools represents practically the only hope of discovering what the program is actually doing. Several things must be discussed to provide an understanding of the connections between overall system description and the actual happenings in the computer at execution time. These connections, or transformations that are made between the overall program and what is actually being done by the computer, are becoming increasingly complex. There seems to be no reason to expect that they will not continue to become more complex, perhaps even at an accelerated rate. The general software and hardware construction of systems suggests that the use of large machines is going to continue to grow. Their economies of scale are already sufficiently impressive that larger numbers of larger machines will probably continue to be developed for a considerable period of time, in spite of the strong rise in minicomputer use in the last few years.

It seems fair to predict that a program as massive and complex as the average DMS is likely to gravitate toward the larger and more complex computers. This would be true even if the economies of scale of large computers were the only factors that affected this evolution. In fact, there is another factor that tends to encourage even more strongly the use of very large and complex computers. The DMS deals with situations in which a large and complicated collection of facts is created. Whenever a large and complex collection of facts is created, it tends to draw attention from many users in many different places. Therefore, there is a demand to make that collection of facts available to a large number of people either by permitting many users to deal more or less simultaneously with the same collection of facts or replicating the collection.

The factors needed to judge the economy scale of the situation must not be limited to usage alone, i.e., a large number of users simultaneously on one machine versus one user at a time on his own machine, but with many machines active at a time. Another factor is the common data base and the problem of keeping that data base current at many different locations. The problem of updating multiple copies is so severe that it alone will dictate a single-system choice in many cases. The multiple-user situation poses many special problems in the description of whether the right logical things are being done by the DMS. To phrase the problem in its simplest terms: Is the DMS functioning? The three following areas of documentation are those for which the prospects of the creation of automatic execution-time documentation that would be useful in determining how well the DMS is functioning are particularly promising.

One is the fixing of the binding point, the point in processing when the indexes of a body of data in the system are actually connected to that data. When DMS performs a search, it usually does so by following a series of steps. First, search criteria are formulated in some fashion. Then, a process occurs whereby some type of preliminary search is made. The term "preliminary search" denotes an activity that stops short of actually examining the data elements themselves and checking whether they are hits or misses on the particular search involved. This is usually done by using the indexes already in the system, as described in the following example.

An activity occurs that attempts to narrow down the search so that a considerable portion of the total file is somehow excluded from consideration. Thus, when an examination is finally made of the particular elements of data that eventually emerge from the search, the indexes, perhaps on several levels, that point toward the body of data in the system are manipulated continually. Tests are made to find out which data elements may be valid and may satisfy the conditions of the search. Often, the system is designed to postpone until a later time the actual retrieval of the surviving elements of data for the final tests of whether each satisfies the criteria of the search. Some systems delay these final tests until the items are recovered from storage; in other instances, the issue is settled completely by examination of the index. Whether this can be done depends, of course, on how the indexes are generated. Whatever is done, the programmer, at debugging time, needs to be able to know the binding point.

From a total processing point of view, it is generally advantageous to delay the binding point as long as possible because to find an item often involves several levels of index lookup before a particular physical location on a disk is found and read. This tends to be a relatively time-consuming operation. Moreover, the item itself is generally much bulkier than its corresponding index entries in a well-conceived system. Therefore, a procedure with a delayed binding point tends to reduce input/output (I/O) load on the computer. In any case, it will have an important effect on what the machine is doing and therefore, will be found in working memory whenever a programmer requests a dump. He must understand this to interpret the dump, to see whether his search is proceeding in the specified way. In other words, he needs to see whether the DMS is functioning correctly. This is one area in which some kind of automatic documentation is needed for some indication of how binding points are being established and when the binding points occur in DMS processing.

The fourth area has to do with things that are concealed from the programmer while he is writing the program. The programmer need not worry about these points as long as the system is functioning properly. Many systems try to optimize processes at execution time. Whatever optimization the system applies to a specified search or to any other process that is specified by the programmer, it will affect the dumps that the programmer sees in case of system malfunction. Optimization results from an effort to make things convenient for the computer at execution time, but it causes the computer to execute steps that are somewhat different from the ones that were actually stated by the programmer. This process of optimization is admirable as long as everything in the optimization logic has been checked out and is valid, but when one is trying to certify that what the program is doing agrees with what it is supposed to do, sometimes difficulties result.

Typically, a DMS causes processes to be data directed. That is, drastically different things happen when different values are actually manipulated by the system at execution time. Systems generally are put into use before every conceivable combination of paths programmed into the system has been exercised and tested. Therefore, it is important for the user to have some way of discovering what the optimizers actually did to his original code. He must know this before he can honestly and sensibly judge whether an apparent malfunction is his mistake or a system error that needs to be corrected. Documentation suitable for this needs to be improved in DMS's.

The final problem area to be discussed is the need for system-supported traces that are reflective of the steps that are followed. These are needed both from the system point of view, so that, for example, the man who is working for the system operator and trying to understand what the computer is doing can proceed in a step-by-step fashion, and from the user point of view, so that the activities relevant to a given user's program can be isolated for review even though his work may be interspersed with that of many other users. All these things need to be documented for the user in a form that is sufficiently detailed to be useful to him in understanding what the system did with his program and, at the same time, general enough that he is not buried by a mass of paper. One of the things that makes this a muddy area is that, clearly, the internal design of the DMS affects the values of a trace. The system features may affect what is worth showing.

For example, one particular system being checked by Computer Sciences Corp. (CSC) has been built with a series of processing modules that are programmed in such a way that there are queues going into and out from each processing routine. All I/O activity is controlled by the overall supervisor, and each routine is self-contained. In a case of this sort, a trace that simply indicates the order in which transactions traverse these individual modules of procedure, all of which have to run all the way to conclusion because of the rules governing the system, can be readily automated, but no more is really needed to permit a user to understand the validity or the lack of validity of what the computer did to his program. It also will facilitate his comparison of what the system actually did with what he programmed. This will greatly assist him in deciding whether he made a mistake or there is something basically wrong with the way the system is performing.

In summary, the modern form of DMS, with the discipline that it enforces by separating the data and the process, often creates a situation in which the process is specified more in a manner that is similar to that employed with decision tables. That is, the details of the steps are sometimes of interest to the programmer, but many times they are not. In cases for which the steps are not of interest to the programmer, it is probably safe to say that, from program to execution, present-day systems provide very little aid in the discovery of the connection, or mapping, and the automatic documentation usually produced by such systems to tie together the things that the programmer does when writing his program and the things that the machine does when executing the program is inadequate to support a proper check of system operation. It seems that this is an area in which automatic documentation can probably make an important contribution, but so little has been done that immediate and important progress can be made whenever the profession mounts a serious effort.

DISCUSSION

MEMBER OF THE AUDIENCE: Would you say a little more about binding points?

SWIFT: Let me use an analogy. Consider the two ways in which algebraic equations are solved. With one approach, values are chosen and substituted into the equation at the start, and the equation is solved numerically. Therefore, the binding point, the point at which the connection is made between the definition of a variable and its value, occurs at the beginning of the solution procedure. If the other approach, the algebraic solution of the equation, is used instead, the binding point occurs at the end of the procedure.

Obviously, what a given system does in developing a binding point is important to our understanding of what ought to be in a given space and core at a given time. Therefore, in a certain sense, it is really a debugging tool.

AUTOFLOW ENHANCEMENTS FOR DOCUMENTATION AND MAINTENANCE OF SCIENTIFIC APPLICATIONS

Martin A. Goetz
Applied Data Research, Inc.

Most documentation of computer programs can be summed up in the phrase, "Even when it's good, it's bad." Management may occasionally give documentation token priority, but programmers seem to give it no priority at all, perhaps because of their training. Programmer training is either formal or informal. In formal training courses, documentation is usually not a standard part of the curriculum; in informal or on-the-job training, it is usually not even mentioned. This lack of training is a basic reason for the problem of documentation, a problem that is compounded whenever management deemphasizes program documentation simply because past experience has shown that what had been produced was generally ineffective.

The chief reason that documentation is so poor may be that it has been considered a manual process when it should have been considered a computer problem. Certainly, no one considers compiling a manual process today, although, years ago, compiler functions were performed manually.

The need for documentation seems to be obvious. The primary concerns of both managers and programmers are program productivity, debugging, flexibility, integration, and reliability. Good documentation helps to fulfill these purposes; poor documentation, on the other hand, does not. Any organization can obtain good documentation, either manual or automatic, if it concentrates on program organization rules; programming standards, including the naming of tagged lines, proper commentary, modular programming, and restrictions in the use of certain programming techniques; program monitoring and security, including systematic recording of changes in programs, systematic recording of reasons for changes, and protection of programs; technical overviews of programs (using tape recordings, if preferred); and parallel development of programs and documentation.

Program organization rules are important because, although good programmers have an organized approach to writing programs, they, unfortunately, usually develop styles of their own. Rarely will two programmers use the same organization. Because a programmer does not work on a program forever, it is obvious that organization should not be permitted to suffer from the idiosyncrasies of the individual programmer. The same can be said for programming standards, which, by definition, can be effective only if they are both universally published and observed.

If programmers followed consistent program organization rules and programming standards, much of today's documentation problem would not have arisen. The computer industry is almost 20 years old; it should stop philosophizing about what ought to be and resolve this unsatisfactory situation.

Only automated documentation of programs offers any hope for realizing what may be called "accurate" program documentation. This paper will discuss how to improve automated documentation and, specifically, how the AUTOFLOW system can be enhanced to provide acceptable levels of documentation.

Given that programmers may cooperate only to a limited extent in documenting their programs and that computer programs can be developed to generate information that could not be produced manually, the following three elements are essential for an integrated documentation system within the framework of today's data processing environment:

(1) Logical analysis or graphic dissection of a program
(2) History and control of programs
(3) An understanding of the program

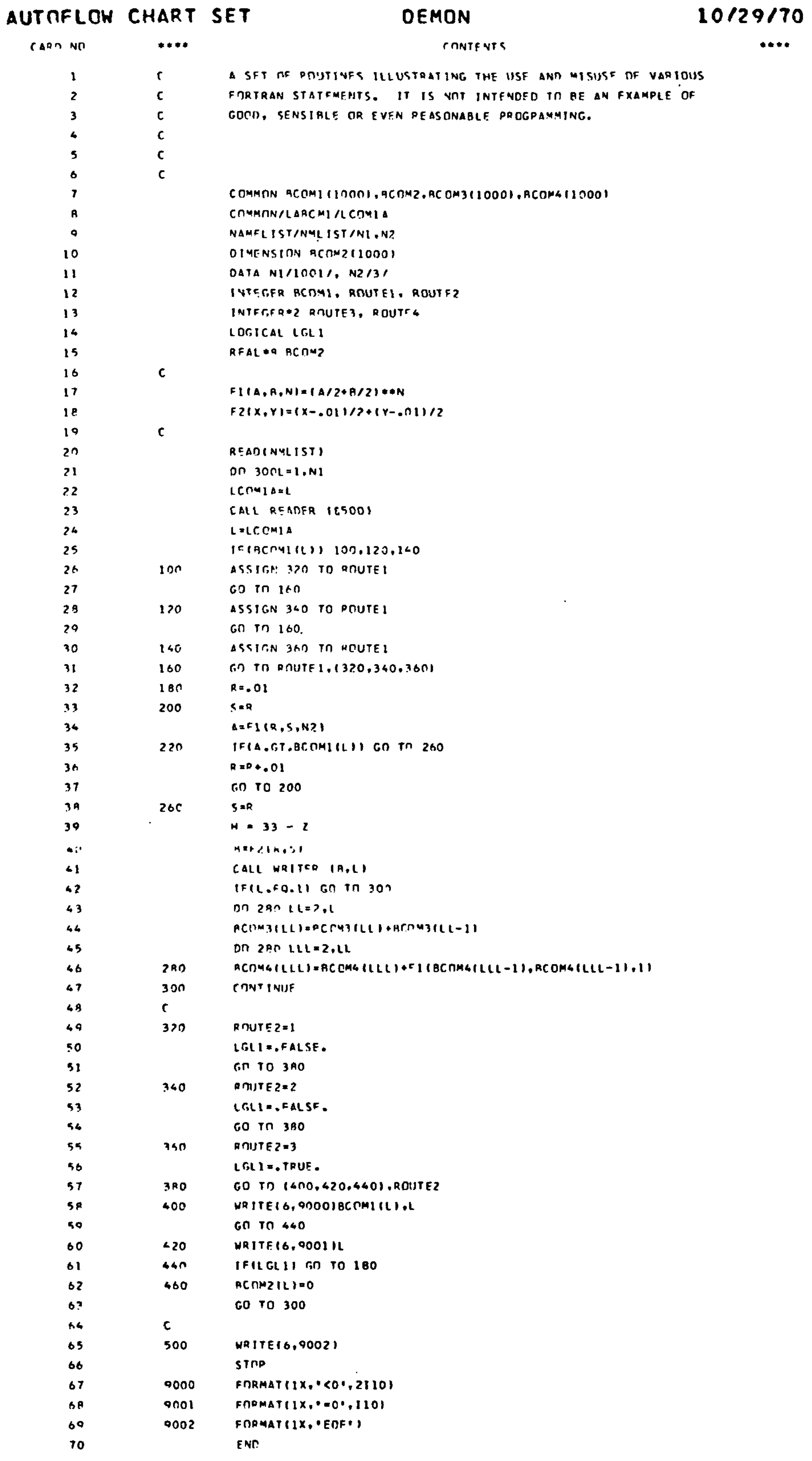
A flowchart produced by AUTOFLOW is much more meaningful than one that has been produced manually. These logical flowcharts are accurate, present complete references between all transfer points, and graphically portray the logical flow by automatic rearrangement of those segments of the program that interact. Figure 1 is an example of a two-dimensional AUTOFLOW flowchart from a FORTRAN program.

The number and type of cross-referenced reports produced by AUTOFLOW depend on the source language being used. For COBOL, AUTOFLOW can produce four special reports: procedure division summary, data name cross-reference listing, data division index, and data record map. For PL/I, four special reports are produced: on-unit action blocks, label-assignment cross-reference, duplicate declaration map, and condition prefix map. For FORTRAN, the one special report is the nonprocedural statements listing. Other special reports for FORTRAN could be produced by AUTOFLOW and would be of great value. Figures 2 through 10 are hypothetical reports that could be produced from a FORTRAN program by systems such as AUTOFLOW.

Figure 2 illustrates the header information that is common to all reports. The information includes the general title, FORTRAN analysis report; the user name, e.g., Goddard Space Flight Center; and the system. The run time for the analysis and the data are also presented. The report itself is essentially a listing of the local variables used by the program. The information presented is the mnemonic label, the type of variable, the definition of the variable, the line number where it is defined, the type and value of the definition, and then the references made by other statements in the FORTRAN source program to the local variable.

References in all reports consist of the source line number and, in parentheses, the AUTOFLOW page and box number. The variable labels in the first column are sorted alphanumerically. The label types are standard for IBM FORTRAN (integer 2, integer 4, real 4, real 8, logical, etc.). The DECLARATIONS column specifies where and how the variable is defined (i.e., through a data statement or an equivalence statement). If the variable is defined by a data statement, the value of the definition will be shown. Doubly-defined variables would be indicated by the notation DD in the definition area.

Figure 3, a cross-reference of statement numbers, lists only those statements that can be referenced by other statements within a program, i.e., statements with statement numbers. The appropriate line number, flowchart location, and type of statement (e.g., format,



```
AUTOFLOW CHART SET                DEMON                10/29/70
 CARD NO      ****                  CONTENTS                  ****

    1         C       A SET OF ROUTINES ILLUSTRATING THE USE AND MISUSE OF VARIOUS
    2         C       FORTRAN STATEMENTS.  IT IS NOT INTENDED TO BE AN EXAMPLE OF
    3         C       GOOD, SENSIBLE OR EVEN REASONABLE PROGRAMMING.
    4         C
    5         C
    6         C
    7                 COMMON BCOM1(1000),BCOM2,BCOM3(1000),BCOM4(1000)
    8                 COMMON/LABCM1/LCOM1A
    9                 NAMELIST/NMLIST/N1,N2
   10                 DIMENSION BCOM2(1000)
   11                 DATA N1/1001/, N2/3/
   12                 INTEGER BCOM1, ROUTE1, ROUTE2
   13                 INTEGER*2 ROUTE3, ROUTE4
   14                 LOGICAL LGL1
   15                 REAL*8 BCOM2
   16         C
   17                 F1(A,B,N)=(A/2+B/2)**N
   18                 F2(X,Y)=(X-.01)/2+(Y-.01)/2
   19         C
   20                 READ(NMLIST)
   21                 DO 300L=1,N1
   22                 LCOM1A=L
   23                 CALL READER (&500)
   24                 L=LCOM1A
   25                 IF(BCOM1(L)) 100,120,140
   26         100     ASSIGN 320 TO ROUTE1
   27                 GO TO 160
   28         120     ASSIGN 340 TO ROUTE1
   29                 GO TO 160.
   30         140     ASSIGN 360 TO ROUTE1
   31         160     GO TO ROUTE1,(320,340,360)
   32         180     R=.01
   33         200     S=R
   34                 A=F1(R,S,N2)
   35         220     IF(A.GT.BCOM1(L)) GO TO 260
   36                 R=R+.01
   37                 GO TO 200
   38         260     S=R
   39                 H = 33 - Z
   40                 B=F2(R,S)
   41                 CALL WRITER (B,L)
   42                 IF(L.EQ.1) GO TO 300
   43                 DO 280 LL=2,L
   44                 BCOM3(LL)=BCOM3(LL)+BCOM3(LL-1)
   45                 DO 280 LLL=2,LL
   46         280     BCOM4(LLL)=BCOM4(LLL)+F1(BCOM4(LLL-1),BCOM4(LLL-1),1)
   47         300     CONTINUE
   48         C
   49         320     ROUTE2=1
   50                 LGL1=.FALSE.
   51                 GO TO 380
   52         340     ROUTE2=2
   53                 LGL1=.FALSE.
   54                 GO TO 380
   55         360     ROUTE2=3
   56                 LGL1=.TRUE.
   57         380     GO TO (400,420,440),ROUTE2
   58         400     WRITE(6,9000)BCOM1(L),L
   59                 GO TO 440
   60         420     WRITE(6,9001)L
   61         440     IF(LGL1) GO TO 180
   62         460     BCOM2(L)=0
   63                 GO TO 300
   64         C
   65         500     WRITE(6,9002)
   66                 STOP
   67         9000    FORMAT(1X,'<0',2I10)
   68         9001    FORMAT(1X,'=0',I10)
   69         9002    FORMAT(1X,'EOF')
   70                 END
```

Figure 1.–AUTOFLOW flowchart for FORTRAN program.

```
10/29/70          INPUT LISTING                              AUTOFLOW CHART SET - DEMON

FORTRAN MODULE       (NAMSO,LIST)

  CARD NO          ****                            CONTENTS                                   ****

      1                          SUBROUTINE READER (*)
      2            C
      3                          COMMON BCOM1(1000),BCOM2(1000),BCOM3(1000),BCOM4
      4                          DIMENSION BCOM4(1000),EQ1(1000),EQ2(1000),XQ1(10)
      5                          COMMON/LABCM1/LCOM1A
      6                          COMMON/LABCM2/LCOM2A
      7                          INTEGER EQ1,XQ2
      8                          EQUIVALENCE (EQ1,BCOM1)
      9                          REAL*8BCOM2,XQ2
     10            C
     11                          READ(5,9000,END=500)EQ1(LCOM1A)
     12                          X=EQ1(L1COMA)
     13                          CALL WRITER (X,LCOM1A)
     14                          LCOM1A=LCOM2A
     15                          RETURN
     16                          F=14
     17            500           RETURN1
     18            9000          FORMAT(58X,I5)
     19                          END
```

```
10/29/70          INPUT LISTING                              AUTOFLOW CHART SET - DEMON

FORTRAN MODULE       (NAMSO,LIST)

  CARD NO          ****                            CONTENTS                                   ****

      1                          SUBROUTINE WRITER (X,J)
      2            C
      3                          COMMON BCOM1(1000),BCOM2(1000),BCOM3(1000),BCOM4(1000)
      4                          COMMON/LABCM2/LCOM2A,LCOM2B
      5                          REAL*8 X
      6            C
      7                          WRITE(6,9000)X,J
      8                          G = 23 + YY
      9                          IF(LCOM2A.GT.2)LCOM2A=LCOM2A-SQRT(3.)
     10                          RETURN
     11                          F=21
     12            9000          FORMAT('0',F20.4,I10)
     13                          END
```

```
10/29/70        PROCEDURAL STATEMENT LABEL INDEX        AUTOFLOW CHART SET - DEMON                               PAGE    1

          PG.BX    NAME            PG.BX    NAME            PG.BX    NAME            PG.BX    NAME            PG.BX    NAME

           2.08  100                3.01  180                3.11  280                3.17  360                3.23  440
           2.01  120                3.02  200                3.14  300                3.18  380                3.24  460
           2.09  140                3.03  220                3.15  320                3.19  400                3.25  500
           2.10  160                3.05  260                3.16  340                3.21  420
```

```
10/29/70        PROCEDURAL STATEMENT LABEL INDEX        AUTOFLOW CHART SET - DEMON                               PAGE    2

          PG.BX    NAME            PG.BX    NAME            PG.BX    NAME            PG.BX    NAME            PG.BX    NAME

           5.01  READER             5.09  500
```

```
10/29/70        PROCEDURAL STATEMENT LABEL INDEX        AUTOFLOW CHART SET - DEMON                               PAGE    3

          PG.BX    NAME            PG.BX    NAME            PG.BX    NAME            PG.BX    NAME            PG.BX    NAME

           7.01  WRITER
```

Figure 1 (continued).–AUTOFLOW flowchart for FORTRAN program.

```
10/29/70        TABLE OF CONTENTS AND REFERENCES        AUTOFLOW CHART SET - DEMON                                PAGE   1
CARD ID   PAGE/BOX    NAME                              REFERENCES  (SOURCE SEQUENCE NO. AND PAGE/BOX)


FORTRAN MODULE


CHART TITLE - INTRODUCTORY COMMENTS


CHART TITLE - PROCEDURES

(000028)     2.01  120          (000025)   2.07
(000022)     2.04               (000047)   3.14
(000026)     2.08  100
(000030)     2.09  140          (000025)   2.07
(000031)     2.10  160          (000029)   2.01     (000027)   2.08
(000032)     3.01  180          (000061)   3.23
(000033)     3.02  200          (000037)   3.04
(000035)     3.03  220
(000039)     3.05  260          (000035)   3.03
(000044)     3.09               (000046)   3.13
(000046)     3.11  280
(000046)     3.11               (000046)   3.12
(000047)     3.14  300          (000042)   3.07     (000063)   3.24
(000049)     3.15  320          (000031)   2.10
(000052)     3.16  340          (000031)   2.10
(000055)     3.17  360          (000[illegible]   [illegible].10
(000057)     3.18  380          (000[illegible]   3.15     (000054)   3.16
(000058)     3.19  400          (000057)   3.18
(000060)     3.21  420          (000057)   3.18
(000061)     3.23  440          (000057)   3.18     (000059)   3.20
(000062)     3.24  460
(000065)     3.25  500          (000023)   2.05

CHART TITLE - NON-PROCEDURAL STATEMENTS


FORTRAN MODULE


CHART TITLE - SUBROUTINE  READER(*)

(000011)     5.01  READER       (000023)   2.05-X
(000017)     5.09  500          (000011)   5.03

CHART TITLE - NON-PROCEDURAL STATEMENTS


FORTRAN MODULE


CHART TITLE - SUBROUTINE  WRITER(X,J)

(000007)     7.01  WRITER       (000041)   3.06-X   (000013)   5.05-X
(000010)     7.06               (000009)   7.04

CHART TITLE - NON-PROCEDURAL STATEMENTS
```

Figure 1 (continued).–AUTOFLOW flowchart for FORTRAN program.

```
10/29/70                              AUTOFLOW CHART SET - DEMON                              PAGE  01

 CHART TITLE - INTRODUCTORY COMMENTS


                      A SET OF ROUTINES ILLUSTRATING THE USE AND MISUSE OF VARIOUS
                      FORTRAN STATEMENTS.  IT IS NOT INTENDED TO BE AN EXAMPLE OF
                      GOOD, SENSIBLE OR EVEN REASONABLE PROGRAMMING.
```

```
10/29/70                              AUTOFLOW CHART SET - DEMON                              PAGE  02

 CHART TITLE - PROCEDURES
```

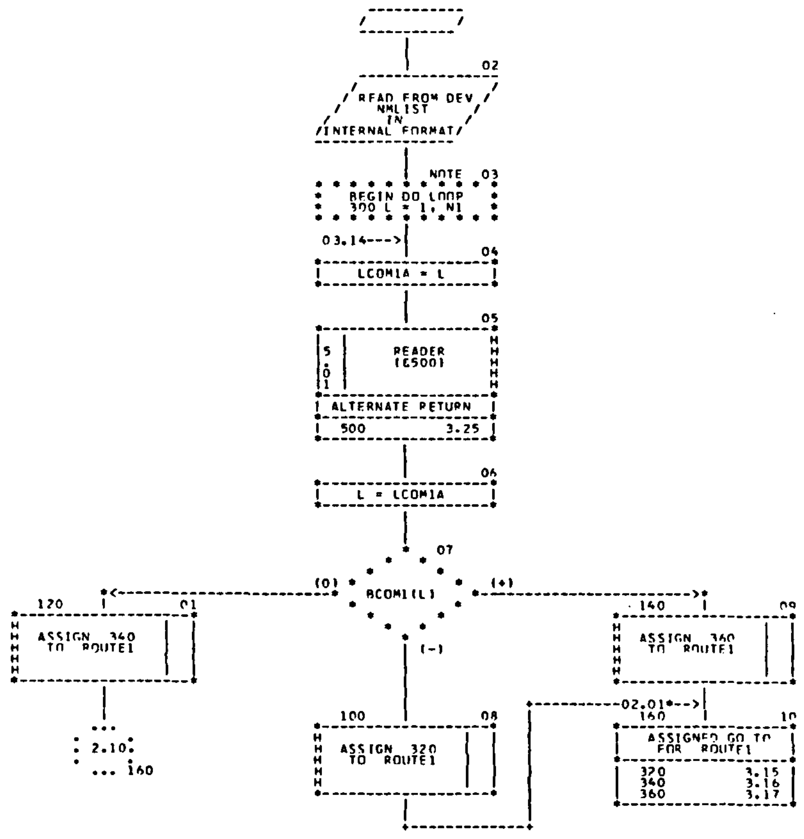


Figure 1 (continued).–AUTOFLOW flowchart for FORTRAN program.

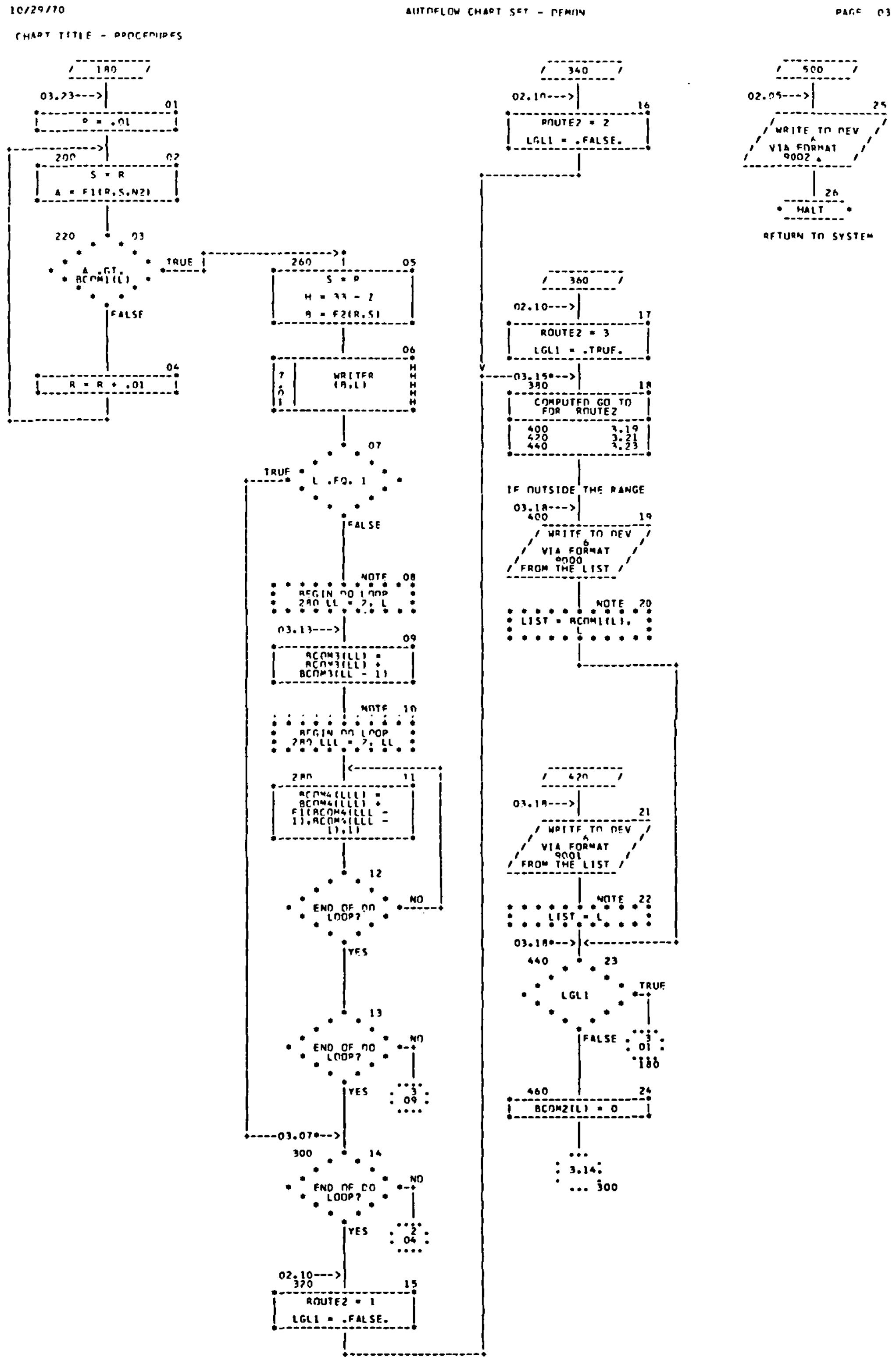


Figure 1 (continued).–AUTOFLOW flowchart for FORTRAN program.

```
10/29/70                              AUTOFLOW CHART SET - DEMON                              PAGE - 04

 CHART TITLE - NON-PROCEDURAL STATEMENTS


                    COMMON BCOM1(1000),BCOM2,BCOM3(1000),BCOM4(1000)
                    COMMON/LABCM1/LCOM1A
                    NAMELIST/NMLIST/N1,N2
                    DIMENSION BCOM2(1000)
                    DATA N1/1001/, N2/3/
                    INTEGER BCOM1, ROUTE1, ROUTE2
                    INTEGER*2 ROUTE3, ROUTE4
                    LOGICAL LGL1
                    REAL*8 BCOM2
                  STATEMENT FUNCTION DEFINITION:     F1(A,B,N)=(A/2+B/2)**N
                  STATEMENT FUNCTION DEFINITION:     F2(X,Y)=(X-.01)/2+(Y-.01)/2
          9000      FORMAT(1X,'<0',2I10)
          9001      FORMAT(1X,'=0',I10)
          9002      FORMAT(1X,'EOF')


   10/29/70                            AUTOFLOW CHART SET - DEMON                            PAGE - 05

  CHART TITLE - SUBROUTINE  READER(*)
```

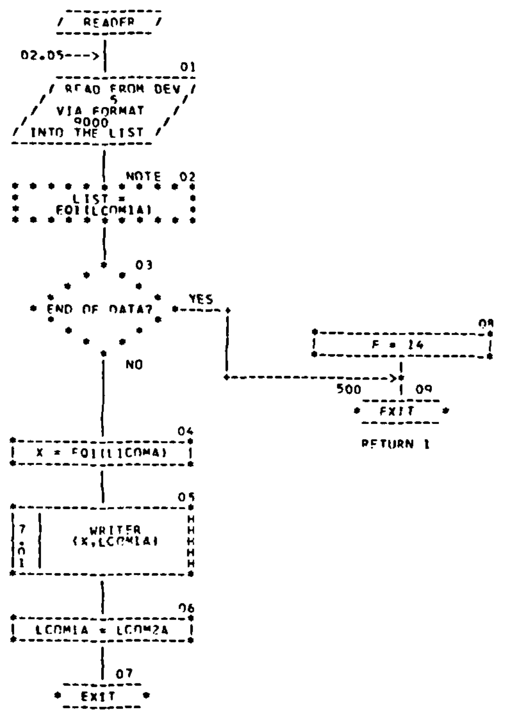
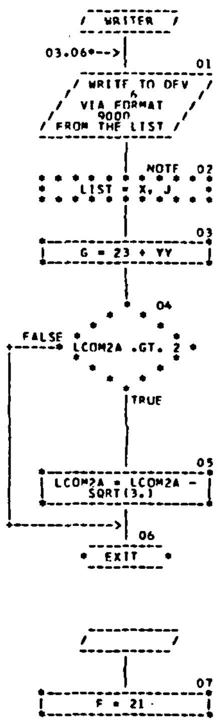


Figure 1 (continued).–AUTOFLOW flowchart for FORTRAN program.

```
10/29/70                                  AUTOFLOW CHART SET - DEMON                              PAGE  06

 CHART TITLE - NON-PROCEDURAL STATEMENTS


                              COMMON BCOM1(1000),BCOM2(1000),BCOM3(1000),BCOM4
                              DIMENSION BCOM4(1000),EQ1(1000),EQ2(1000),XQ1(10)
                              COMMON/LABCM1/LCOM1A
                              COMMON/LABCM2/LCOM2A
                              INTEGER EQ1,XQ2
                              EQUIVALENCE (EQ1,BCOM1)
                              REAL*8BCOM2,XQ2
                    9000      FORMAT(58X,I5)
```

```
10/29/70                                  AUTOFLOW CHART SET - DEMON                              PAGE  07

 CHART TITLE - SUBROUTINE  WRITER(X,J)
```

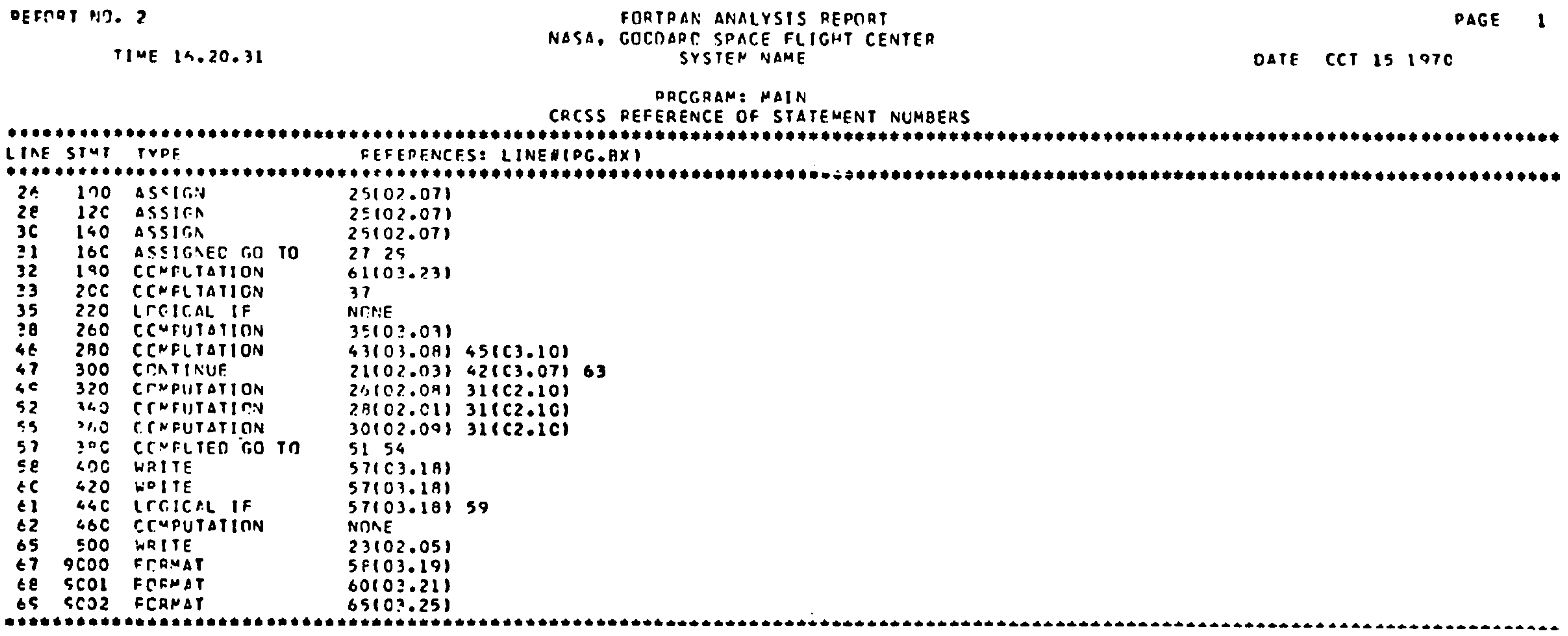


```
10/29/70                                  AUTOFLOW CHART SET - DEMON                              PAGE  08

 CHART TITLE - NON-PROCEDURAL STATEMENTS


                              COMMON BCOM1(1000),BCOM2(1000),BCOM3(1000),BCOM4(1000)
                              COMMON/LABCM2/LCOM2A,LCOM2B
                              REAL*8 X
                    9000      FORMAT('0',F20.4,I10)
```

Figure 1 (concluded).–AUTOFLOW flowchart for FORTRAN program.

```
REPORT NO. 1                                 FORTRAN ANALYSIS REPORT                                                 PAGE   1
                                       NASA, GODDARD SPACE FLIGHT CENTER
         TIME  16.20.31                           SYSTEM NAME                                            DATE  OCT 15 1970

                                                PROGRAM: MAIN
                                       LOCAL VARIABLE REPORT BY PROGRAM
********************************************************************************************************************************
  LABEL     TYPE                                             APPEARANCES: LINE#(PG.BX)
                 *    DECLARATIONS    *                 ASSIGNMENTS                 *                 REFERENCES
********************************************************************************************************************************
A         REAL*4  *                   * 34(03.02)                                   * 35(03.03)
B         REAL*4  *                   * 40(03.05)                                   * 41(03.06)
H         REAL*4  *                   * 39(03.05)                                   * NONE
L         INT*4   *                   * 21(02.03) 24(02.06)                         * 22(02.04) 25(02.07) 35(03.03) 41(03.06) 42(03.07)
                  *                   *                                             * 43(03.08) 58(03.19) 60(03.21) 62(03.24)
LGL1      LOGIC*4 * 14                * 50(03.15) 53(03.16) 56(03.17)               * 61(03.23)
LL        INT*4   *                   * 43(03.08)                                   * 44(03.09) 45(03.10)
LLL       INT*4   *                   * 45(03.10)                                   * 46(03.11)
NMLIST    NLIST   * 09                * 20(02.02)                                   * NONE
N1        INT*4   * 11  DATA     1001 * 11                                          * 21(02.03)
N2        INT*4   * 11  DATA        3 * 11 36                                       * 34(03.02)
R         REAL*4  *                   * 32(03.01) 36(03.04)                         * 33(03.02) 34(03.02) 36(03.04) 38(03.05) 40(03.05)
ROUTE1    INT*4   * 12                * 26(02.08) 28(02.01) 30(02.09)               * 31(02.10)
ROUTE2    INT*4   * 11                * 49(03.15) 52(03.16) 55(03.17)               * 57(03.18)
ROUTE3    INT*2   * 13                *                                             * NONE
ROUTE4    INT*2   * 13                *                                             * NONE
S         REAL*4  *                   * 33(03.02) 38(03.05)                         * 34(03.02) 40(03.05)
Z         REAL*4  * UNDEFINED         * UNDEFINED                                   * 39(03.05)
********************************************************************************************************************************
```

Figure 2.–Header information.



```
REPORT NO. 2                                FORTRAN ANALYSIS REPORT                                              PAGE   1
                                        NASA, GODDARD SPACE FLIGHT CENTER
          TIME 14.20.31                            SYSTEM NAME                                          DATE  OCT 15 1970

                                                  PROGRAM: MAIN
                                        CROSS REFERENCE OF STATEMENT NUMBERS
**************************************************************************************************************************
LINE STMT  TYPE                  REFERENCES: LINE#(PG.BX)
**************************************************************************************************************************
 26   100  ASSIGN               25(02.07)
 28   120  ASSIGN               25(02.07)
 30   140  ASSIGN               25(02.07)
 31   160  ASSIGNED GO TO       27 29
 32   180  COMPUTATION          61(03.23)
 33   200  COMPUTATION          37
 35   220  LOGICAL IF           NONE
 38   260  COMPUTATION          35(03.03)
 46   280  COMPUTATION          43(03.08) 45(03.10)
 47   300  CONTINUE             21(02.03) 42(03.07) 63
 49   320  COMPUTATION          26(02.08) 31(02.10)
 52   340  COMPUTATION          28(02.01) 31(02.10)
 55   360  COMPUTATION          30(02.09) 31(02.10)
 57   380  COMPUTED GO TO       51 54
 58   400  WRITE                57(03.18)
 60   420  WRITE                57(03.18)
 61   440  LOGICAL IF           57(03.18) 59
 62   460  COMPUTATION          NONE
 65   500  WRITE                23(02.05)
 67  9000  FORMAT               58(03.19)
 68  9001  FORMAT               60(03.21)
 69  9002  FORMAT               65(03.25)
**************************************************************************************************************************
```

Figure 3.–Cross-reference of statement numbers.

computational, or assignment), are specified. Again, all references to each statement number are listed by line number and AUTOFLOW page and box references.

Figure 4 is a cross-referenced listing of global variables used by the specific program that is being analyzed. This report is very similar to the local variable report, except that it lists only those variables that reside in blank or labeled common data areas. The information presented in the report includes the label mnemonic, the type of label, its definition, data used in the label, and all references to the label by other statements in the FORTRAN source program. The label type is broken down not only by data type (integer, real, logical, etc.) but also by the type of common area (whether it is blank common or label common and, if label common, by the mnemonic name of the label common area).

Figure 5 is a summary of all of the variables used in all of the programs input to a single AUTOFLOW run and is similar to the local variable report for a specific FORTRAN program. It contains essentially the same kind of information presented in the local variable report, mnemonic label for a variable, the type of variable, the definition of the data for the variable, and all references to that variable. The unique aspect of this report is that it does not reference only those local program variables that are accessible within a specific program but rather those variables that can be passed between programs through a common data area. In the references column, program identification, line number, and AUTOFLOW page and box number are indicated.

Figure 6 is the program subroutine usage report. This presents the names of subroutines within an individual program, the call parameters that are used by or passed to the subroutine, and any references (by line number and AUTOFLOW page and box number) to that subroutine in the specific FORTRAN program being analyzed. In the call parameter area, the variable name that is being passed to the subroutine and some additional information are found. If a global variable is being passed to a subroutine for its own use, an ampersand is appended to the mnemonic label in the CALL statement. A second type of variable that may be passed is a dummy variable, one that is not directly used by the program. This is a variable that has been passed to the present subroutine by a calling subroutine. A dummy variable is indicated by the pound sign appended to it. A third parameter is a return address, indicated by an asterisk. The call parameter portion of the listing also specifies the levels of all variables that are local to the program.

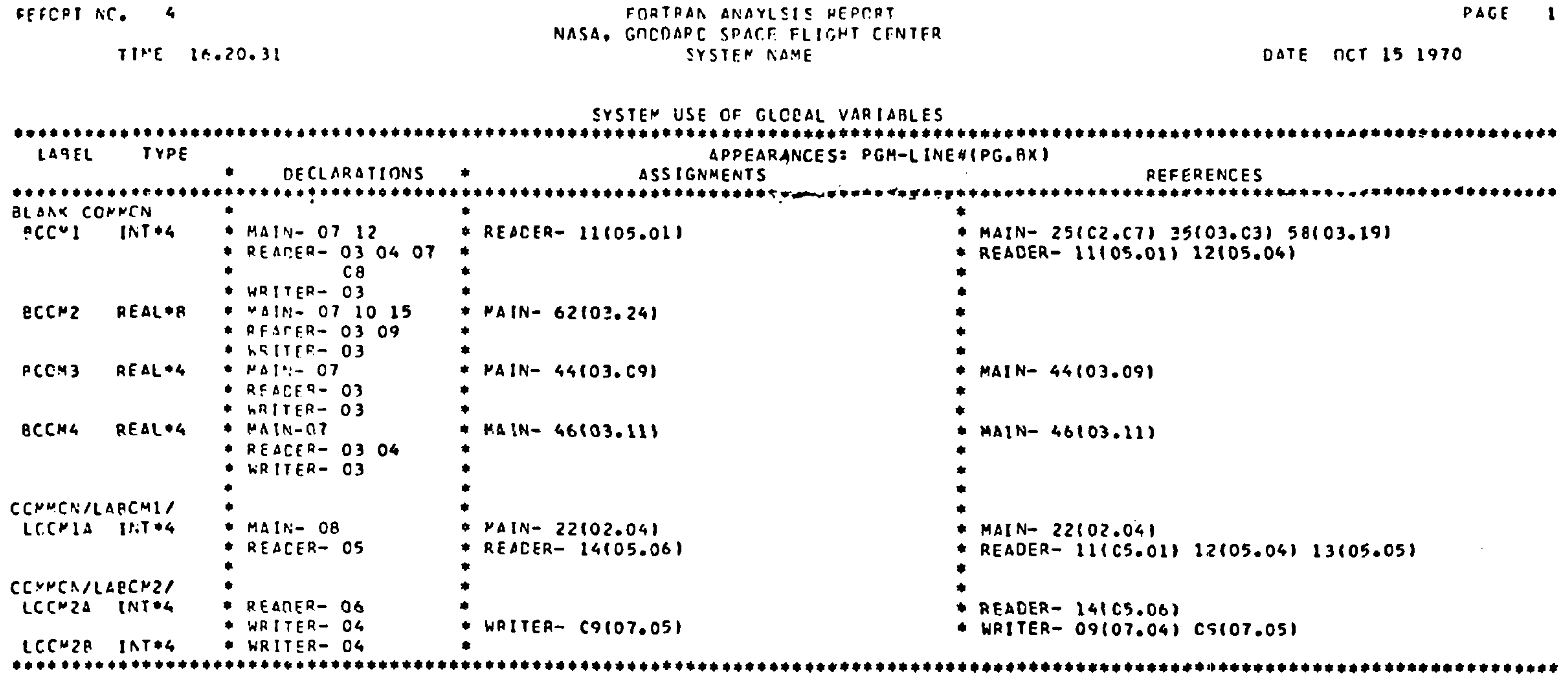
Figure 7, the system subroutine usage report, is very similar to the program subroutine usage report. The name of the program containing the call, the subroutine name, and the local, global, and dummy parameters passed to the called subroutines are specified. The report summarizes all subroutine usage within all program modules processed in a single AUTOFLOW run. Briefly, this listing establishes the hierarchy of subroutine calls among the modules for a given execution.

Figure 8 is the DO loop analysis report for a specific program. This listing indicates the complexity of the DO loop control within the program. The body of the report presents the source and flowchart locations of the start of the loop, the variables used for starting and ending values, and the increment used for the variable counter.

The complexity map, a bar diagram constructed of X's, depicts the logical structure of DO loops in a histogram format. This histogram graphically portrays the nesting effect.

```
REPORT NO.   3                                    FORTRAN ANALYSIS REPORT                                              PAGE   1
                                             NASA, GODDARD SPACE FLIGHT CENTER
       TIME  16.20.31                                   SYSTEM NAME                                        DATE  OCT 15 1970

                                             CROSS REFERENCE OF GLOBAL VARIABLE USE BY PROGRAM
*****************************************************************************************************************************
                                                              PROGRAM: MAIN
  LABEL     TYPE                                                  APPEARANCES: LINE#(PG.BX)
                     *    DECLARATIONS  *                   ASSIGNMENTS                  *                REFERENCES
*****************************************************************************************************************************
BLANK COMMON         *                  *                                                *
 BCOM1    INT*4      * 07 12            *                                                * 25(02.07) 35(03.03) 58(03.19)
 BCOM2    REAL*8     * 07 10 15         * 62(03.24)                                      * NONE
 BCOM3    REAL*4     * 07               * 44(03.09)                                      * 44(03.09)
 BCOM4    REAL*4     * 07               * 46703.11)                                      * 46(03.11)
                     *                  *                                                *
COMMON/LABCM1/       *                  *                                                *
 LCOM1A   INT*4      * 08               * 22(02.04)                                      * 24(02.06)
*****************************************************************************************************************************


*****************************************************************************************************************************
                                                              PROGRAM: READER
  LABEL     TYPE                                                  APPEARANCES: LINE#(PG.BX)
                     *    DECLARATIONS  *                   ASSIGNMENTS                  *                REFERENCES
*****************************************************************************************************************************
BLANK COMMON         *                  *                                                *
 BCOM1               * 03 04 07 08      * 11(05.01)                                      * 11(05.01) 12(05.04)
 BCOM2               * 03 09            *                                                * NONE
 BCOM3               * 03               *                                                * NONE
 BCOM4               * 03 04            *                                                * NONE
                     *                  *                                                *
COMMON/LABCM1/       *                  *                                                *
 LCOM1A              * 05               * 14(05.06)                                      * 11(05.01) 12(05.04) 13(05.05)
                     *                  *                                                *
COMMON/LABCM1/       *                  *                                                *
 LCOM2A              * 06               *                                                * 14(05.06)
*****************************************************************************************************************************


*****************************************************************************************************************************
                                                              PROGRAM: WRITER
  LABEL     TYPE                                                  APPEARANCES: LINE#(PG.BX)
                     *    DECLARATIONS  *                   ASSIGNMENTS                  *                REFERENCES
*****************************************************************************************************************************
BLANK COMMON         *                  *                                                *
 BCOM1               * 03               *                                                * NONE
 BCOM2               * 03               *                                                * NONE
 BCOM3               * 03               *                                                * NONE
 BCOM4               * 03               *                                                * NONE
                     *                  *                                                *
COMMON/LABCM1/       *                  *                                                *
 LCOM2A              * 04               * 09(07.05)                                      * 09(07.04) 09(07.05)
 LCOM12              * 04               *                                                *
*****************************************************************************************************************************
```

Figure 4.–Cross-reference of global variables used.



```
REPORT NO.   4                          FORTRAN ANAYLSIS REPORT                                   PAGE   1
                                    NASA, GODDARD SPACE FLIGHT CENTER
        TIME  16.20.31                          SYSTEM NAME                             DATE  OCT 15 1970

                                      SYSTEM USE OF GLOBAL VARIABLES
*****************************************************************************************************************
  LABEL    TYPE                                     APPEARANCES: PGM-LINE#(PG.BX)
                 *    DECLARATIONS   *               ASSIGNMENTS                              REFERENCES
*****************************************************************************************************************
BLANK COMMON     *                   *                                   *
 BCCM1    INT*4  * MAIN- 07 12       * READER- 11(05.01)                 * MAIN- 25(02.07) 35(03.03) 58(03.19)
                 * READER- 03 04 07  *                                   * READER- 11(05.01) 12(05.04)
                 *         08        *                                   *
                 * WRITER- 03        *                                   *
 BCCM2    REAL*8 * MAIN- 07 10 15    * MAIN- 62(03.24)                   *
                 * READER- 03 09     *                                   *
                 * WRITER- 03        *                                   *
 BCCM3    REAL*4 * MAIN- 07          * MAIN- 44(03.09)                   * MAIN- 44(03.09)
                 * READER- 03        *                                   *
                 * WRITER- 03        *                                   *
 BCCM4    REAL*4 * MAIN-07           * MAIN- 46(03.11)                   * MAIN- 46(03.11)
                 * READER- 03 04     *                                   *
                 * WRITER- 03        *                                   *
                 *                   *                                   *
COMMON/LABCM1/   *                   *                                   *
 LCCM1A   INT*4  * MAIN- 08          * MAIN- 22(02.04)                   * MAIN- 22(02.04)
                 * READER- 05        * READER- 14(05.06)                 * READER- 11(05.01) 12(05.04) 13(05.05)
                 *                   *                                   *
COMMON/LABCM2/   *                   *                                   *
 LCCM2A   INT*4  * READER- 06        *                                   * READER- 14(05.06)
                 * WRITER- 04        * WRITER- 09(07.05)                 * WRITER- 09(07.04) 09(07.05)
 LCCM2B   INT*4  * WRITER- 04        *
*****************************************************************************************************************
```

Figure 5.–System use of global variables.

```
REPORT NO     5                                          FORTRAN ANALYSIS REPORT                                          PAGE   1
                                                   NASA, GODDARD SPACE FLIGHT CENTER
       TIME  16.20.31                                          SYSTEM NAME                                       DATE  OCT 15 1970

                                                          SUBROUTINE USE  BY PGM
*******************************************************************************************************************************
                                                            PROGRAM: MAIN
  SUBROUTINE *  CALL PARAMETERS PASSED (@=COMMON VAR, #=DUMMY VAR,*=NON.STD RTURN)* CALLING REFERENCES: LINE(PG.BX)
*******************************************************************************************************************************
             *                                                                      *
  READER     *  (*)                                                                 * 23(02.05)
             *                                                                      *
  WRITER     *  (*,J)                                                               * 41(03.46)
*******************************************************************************************************************************
```

Figure 6.–Program subroutine usage report.

```
REPORT NO.   6                                        FORTRAN ANALYSIS REPORT                                              PAGE   1
                                                NASA, GODDARD SPACE FLIGHT CENTER
        TIME  16.20.31                                     SYSTEM NAME                                          DATE OCT 15 1970

                                                  SYSTEM SUBROUTINE ANALYSIS

**************************************************************************************************************
                                                      SYSTEM SUB CALL ANALYSIS
PGM NAME SUB CALLED PARAMETERS PASSED (@=GLOBAL VAR, #=DUMMY VAR,  *=NON-STD RETURN)
**************************************************************************************************************
MAIN      READER      *
          WRITER       B, L

READER    WRITER       X, @LCOMIA
**************************************************************************************************************


**************************************************************************************************************
                                                  SYSTEM SUBROUTINE USAGE
 NAME    *  CALL PARAMETERS  (@=COMMON VAR, #=DUMMY VAR, *=NON-STC RTRN)* CALLING REFERENCES: PGM-LINE#(PG.BX)
**************************************************************************************************************
READER   *  (*)                                                       * MAIN- 23(02.05)
         *                                                            *
WRITER   *  (X,J)                                                     * READER- 13(05.05) MAIN- 41(03.06)
**************************************************************************************************************
```

Figure 7.–System subroutine usage report.

```
REPORT NO  7                                          FORTRAN ANALYSIS REPORT                                          PAGE   1
                                                  NASA, GODDARD SPACE FLIGHT CENTER
         TIME  14.20.31                                     SYSTEM NAME                                     DATE  OCT 15 1970

                                                   DETAIL OF LOOP  ANALYSIS BY PROGRAM
***************************************************************************************************************************
                                                                 PROGRAM: MAIN
 START          END             LOOP CONTROL            COMPLEXITY MAP
LINE  STMT   LINE  STMT   VARIABLE INIT TEST INCR           LINE
***************************************************************************************************************************

21            47    300    L          1    N1    1        X
31    160                                                 X                               EXIT TO 320,340,360
43            46    280    LL         2    L     1        X X
45            46    280    LLL        2    LL    1        X X X
46    280                                                 X X X
47    300                                                 X



                                                          SUMMARY
                                                    LEVEL      NO OF LOOPS

                                                      1             1

                                                      2             1

                                                      3             1

***************************************************************************************************************************
```

Figure 8.–DO loop analysis report.

Additional information, such as an exit from within a loop to a statement external to the loop, is also shown by the histogram. A nest of three loops is represented by three vertical bars. The longest bar represents the initial DO loop, the next longest represents the second-level loop, and the shortest represents the third-level loop. The second part of this listing is the DO loop analysis summary, which specifies the loop level and the number of loops of different levels used in the program.

Figure 9 is the assigned GO TO analysis by program. This listing presents the sequence, page and box numbers, and statement number of all the assigned GO TO statements in a FORTRAN program. Additionally, variable names used in the branch list for each assigned GO TO are presented. The right side of the report lists all references to particular assigned GO TO statements. If one of the variables in the branch list is not defined within the program, this variable name will be listed with a dollar sign indicator. This is particularly helpful since undefined variables used in assigned GO TO statements will result in unpredictable destinations for the branch. The logic analysis section of this report presents program conditions that are probable program errors (e.g., undefined labels, unreferenced statements, undefined variables, or transfers into a DO loop).

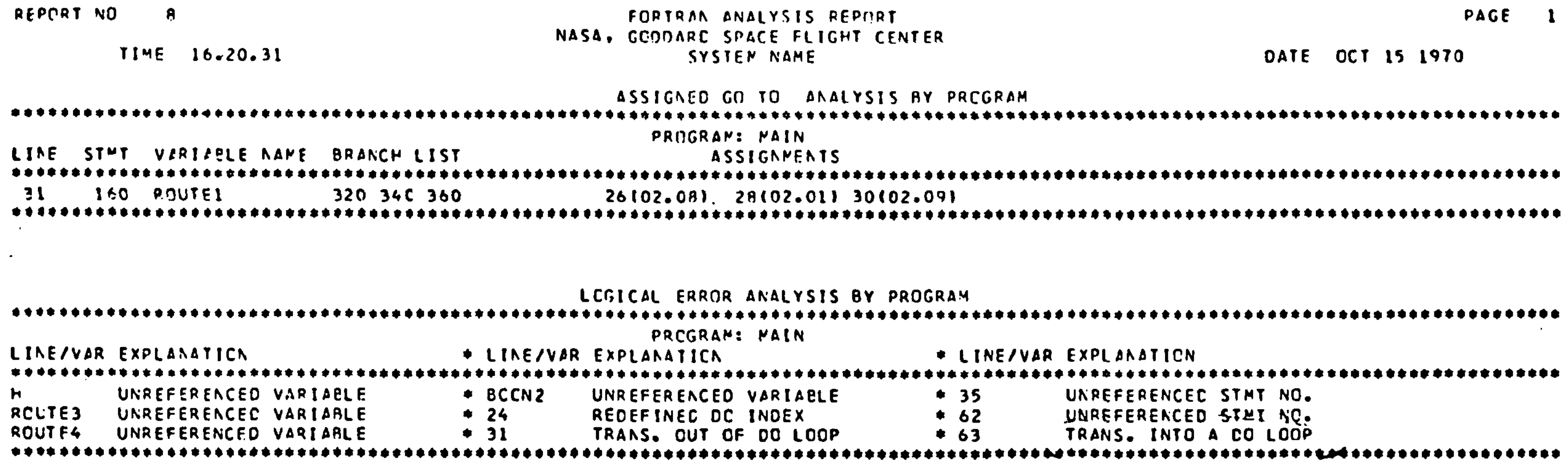
Figure 10 is the statement usage and complexity factor report, which presents a weighted summary of statement types within a program. On the left side of the report is the statement type (such as assigned GO TO, computed GO TO, dimension, value, and computational) and the number of each type within a program. The listing also contains the information needed for the complexity factor analysis. The assigned weight factors and the weighted values automatically assigned to the different types of statements. The user may override the default values and assign his own weighted factors at execution time. The product of the number of statements of a particular type and the weight factor for that type is the usage factor. At the bottom of this report is a summary which shows the total number of statements in the FORTRAN program, the total weight (the sum of all the usage factors), and the program complexity (the computed value of the total weight divided by the total number of statements). Program complexities range from 0.1 to 0.9. A factor of 0.5 would indicate that the program is of average complexity. The complexity factor is a useful guide for effective programmer assignment.

HISTORY AND CONTROL OF PROGRAMS

A program represents a considerable asset to an organization because it is usually costly to develop and is used to control functions within an organization ranging from the performance of simple accounting operations to the control of space flight programs.

Many programs have a life span far in excess of 5 years. A case in point is the IBM 650 program, which was simulated on the IBM 1401 after the IBM 650 was removed. The IBM 1401 is now being simulated on the IBM 360 and will shortly be simulated on the IBM 370. Rumor has it that the IBM 650 program was actually simulating an IBM 604 tabulating function.

Programs survive intact over long periods of time because they are infrequently run and, therefore, not economical to reprogram, or nobody really knows their contents (the fear factor). In general, today's software technology is in such a deplorable condition for



```
REPORT NO    8                                      FORTRAN ANALYSIS REPORT                                           PAGE   1
                                              NASA, GODDARD SPACE FLIGHT CENTER
         TIME  16.20.31                                  SYSTEM NAME                                      DATE  OCT 15 1970

                                                   ASSIGNED GO TO  ANALYSIS BY PROGRAM
***********************************************************************************************************************************
                                                      PROGRAM: MAIN
LINE  STMT  VARIABLE NAME  BRANCH LIST                        ASSIGNMENTS
***********************************************************************************************************************************
 31   160  ROUTE1          320 34C 360           26(02.08), 28(02.01) 30(02.09)
***********************************************************************************************************************************


                                                LOGICAL ERROR ANALYSIS BY PROGRAM
***********************************************************************************************************************************
                                                      PROGRAM: MAIN
LINE/VAR EXPLANATION                  * LINE/VAR EXPLANATION                    * LINE/VAR EXPLANATION
***********************************************************************************************************************************
H        UNREFERENCED VARIABLE        * BCCN2    UNREFERENCED VARIABLE          * 35       UNREFERENCEC STMT NO.
RCUTE3   UNREFERENCEC VARIABLE        * 24       REDEFINEC DC INDEX             * 62       UNREFERENCED STMT NO.
ROUTE4   UNREFERENCED VARIABLE        * 31       TRANS. OUT OF DO LOOP          * 63       TRANS. INTO A DO LOOP
***********************************************************************************************************************************
```

Figure 9.—Assigned GO TO analysis.

```
REPORT NO     9                     FORTRAN ANALYSIS REPORT                                   PAGE   1
                                NASA, GODDARD SPACE FLIGHT CENTER
         TIME   16.20.31                  SYSTEM NAME                               DATE  OCT 15 1970

                                       PROGRAM: MAIN
                                STATEMENT USAGE AND COMPLEXITY FACTORS

                             TYPE              NUMBER     ASSIGNED     USAGE
                                               IN PGM     WT FACTOR    FACTOR

                        ASSIGN GO TO              1          1.0         1.0
                        COMPLEX                   0          0.2          .0
                        COMPUTED GO TO            1          1.0         1.0
                        CONTINUE                  1          0.2          .2
                        DATA                      1          0.5          .5
                        DIMENSION                 1          0.3          .3
                        DO                        3          0.5         1.5
                        EQUIVELENCES              0          0.6          .0
                        FORMAT                    3          0.3          .9
                        FUNCTION DEF.             2          0.7         1.4
                        GO TO                     7          0.9         6.3
                        IF                        4          0.5         2.0
                        INTEGER                   2          0.2          .4
                        LOGICAL                   1          0.2          .2
                        READ                      1          0.1          .1
                        REAL                      1          0.2          .2
                        SUB CALL                  2          0.8         1.6
                        STMT FUNCT CALL           3          0.7         2.1
                        WRITE                     3          0.1          .3
                        ASSIGNMENTS              18          0.1         1.8
                        NAMELIST                  1          0.4          .4

                        .......REPORT SUMMARY.........

                         A. TOTAL STATEMENTS    56

                         B. TOTAL WEIGHT....    22.2

                         C. COMPLEXITY..(B/A)   .40
```

Figure 10.–Statement usage and complexity factor report.

the latter reason. Programs such as The LIBRARIAN, an adjunct to the AUTOFLOW system, are available to monitor program activity; produce histories of changes; retain copies of old versions of programs; protect programs against unauthorized use; and provide complete indexes that give dates of modifications, reasons for changes, and other information necessary for the orderly maintenance of programs and data.

UNDERSTANDING THE PROGRAM

The next questions to be asked concern the function, organization, and reason for organization of a program. All these questions can be answered by "picking the brains" of the programmer and the designer.

Given the aversion of most programmers to documentation, the tape recorder can be a very effective means of obtaining vital information. It is probably much easier for many programmers to sit down and record on a cassette all the details of program development than for them to take the time to write everything down. The taped information can be easily transcribed and converted to a machine-readable form for input to a system such as TEXT EDITOR. This system can be used to produce a finished document for permanent retention as the program history and enables a user to specify format, alter content, and expedite production of hard-copy documentation with a minimum of manual effort. In short, the programmer need only talk about his projects, and a final record of such discussions can be automatically produced.

The final issue that is critical for the overall effectiveness of documentation is whether it actually reflects the current status of program development. Outdated documentation can be only partially useful at best, and totally misleading at worst. The systems discussed, AUTOFLOW, The LIBRARIAN, and TEXT EDITOR, assure all users that the documentation will be not only accurate, standardized, and complete but also timely and readily available whenever needed.

CONCLUSION

In summary, the critical needs in the area of effective program documentation involve the integration of normal programming activities with the requirement for more comprehensive documentation. The ultimate solution to these needs lies in automated documentation systems that can reduce clerical effort on the part of the programmer, provide timely and accurate documentation whenever needed, analyze program design and structure, expedite maintenance and debugging operations, protect source programs from loss or damage, and provide an understanding of the program. Computer programs can do this and can do it better, faster, and more economically.

DISCUSSION

MEMBER OF THE AUDIENCE: I understand that AUTOFLOW is applicable to FORTRAN; is it also applicable to other programming languages?

GOETZ: AUTOFLOW can be applied to all of the major languages in use today, including second-generation programming languages and various types of FORTRAN.

MEMBER OF THE AUDIENCE: To your knowledge, does anyone else employ the tape recorder in the way that you have discussed, and what benefits does it offer to programming personnel?

GOETZ: Although I am certain that it must be used elsewhere, I cannot provide any specific organization names. The technique makes it easier for the programmer to record information. The information generated is actually of better quality than that which would be produced if the programmer were required to write his documentation, since the programmer becomes too self-conscious when he is writing.

MEMBER OF THE AUDIENCE: Do you have any intention of writing a manual describing the entire procedure that could be marketed?

GOETZ: We have no current plans for doing that.

MEMBER OF THE AUDIENCE: You have mentioned that AUTOFLOW is available for several different language systems. Does this diversity also extend to different computers?

GOETZ: AUTOFLOW is not available for many machines; it is available for the Spectra 70 series, the Honeywell series, and the IBM 7090 and 360 series.

MEMBER OF THE AUDIENCE: Is there an extended AUTOFLOW available for the CDC 6600?

GOETZ: No. The AUTOFLOW system is written in assembly language and cannot be transferred between machines. No AUTOFLOW was written for the CDC 6600. We do accept 6600 programs—assembly language and the various FORTRANS, I believe—but the AUTOFLOW system does not operate with them. Also, the extended versions of the FORTRAN analysis are hypothetical systems that have not yet been constructed. The flowcharts and reports used in my paper were manually produced.

MEMBER OF THE AUDIENCE: What use is made of the tape recorder in the development of the user documentation?

GOETZ: The program documentation, providing the internal logic of the program, can best be obtained with the use of the tape recorder, but the user documentation is something quite different. It should be well organized and produced in a more formal way than the program documentation.

MEMBER OF THE AUDIENCE: Do the American National Standards Institute (ANSI) flowchart standards constrain the actual communication of information because of restrictions placed on the size and proportion of symbols and the lack of symbols needed to terminate and then continue a line that is not related to the flow of the data or the logic of the program? Since symbols in modern languages can have as many as 30 characters, the standards, to a certain extent, inhibit communication because the programmer must limit what he says.

GOETZ: Our current standards do not quite conform to ANSI standards. The width of a process box, for instance, must be related to its length, according to ANSI standards, but AUTOFLOW will produce a process box of virtually any size, so it could be 50 or 100 lines long. We are upgrading our system so that it will conform completely to ANSI standards, which will restrict or inhibit somewhat the flowchart produced. The user will then have the option of having ANSI or AUTOFLOW standards.

MEMBER OF THE AUDIENCE: Do you consider the ANSI standards to be adequate or archaic?

GOETZ: We think that they are somewhat archaic, but they are standards, and we are willing to conform. Therefore, we are producing the option.

MEMBER OF THE AUDIENCE: Consider a program that was written without AUTO-FLOW in mind. If the program were then analyzed by AUTOFLOW, which would be the most useful: analysis portion or the flowchart portion?

GOETZ: It would depend upon who would be using the report. For the original programmer, the analysis portion will suffice in many cases. For debugging and making program alterations, the flowchart is especially useful and would probably be a necessary aid if those functions were being performed by someone who was not the original programmer. The level of the programmer's training would also be a consideration.

MEMBER OF THE AUDIENCE: To what extent is AUTOFLOW used to document and maintain itself?

GOETZ: The entire system is written in Assembly language and contains chart codes in the comments portion of the program. By putting these chart codes in the program and considering what the assembly language coding represents, we obtain very good narrative statements and comments. The very low personnel turnover that we have reduces considerably the need for producing flowcharts for maintenance purposes.

N73-19207

THE BELLFLOW SYSTEM

Stephen Pardee
Bell Telephone Laboratories

STANDARDS

One of the primary reasons for the development of BELLFLOW was the need to meet certain Bell System standards of documentation. The Bell System has been documenting designs for a long time and, in addition to trying to observe outside standards, has additional internal standards that must be observed. In the area of flowcharting, most of these additional standards relate to quality and not to symbol or line character size, legibility, titling information, change information, information placement, and format. BELLFLOW had to be able to meet these quality standards. The symbols incorporated in BELLFLOW are essentially the standard flowcharting symbols. (See fig. 1 for a list of the BELLFLOW symbols.)

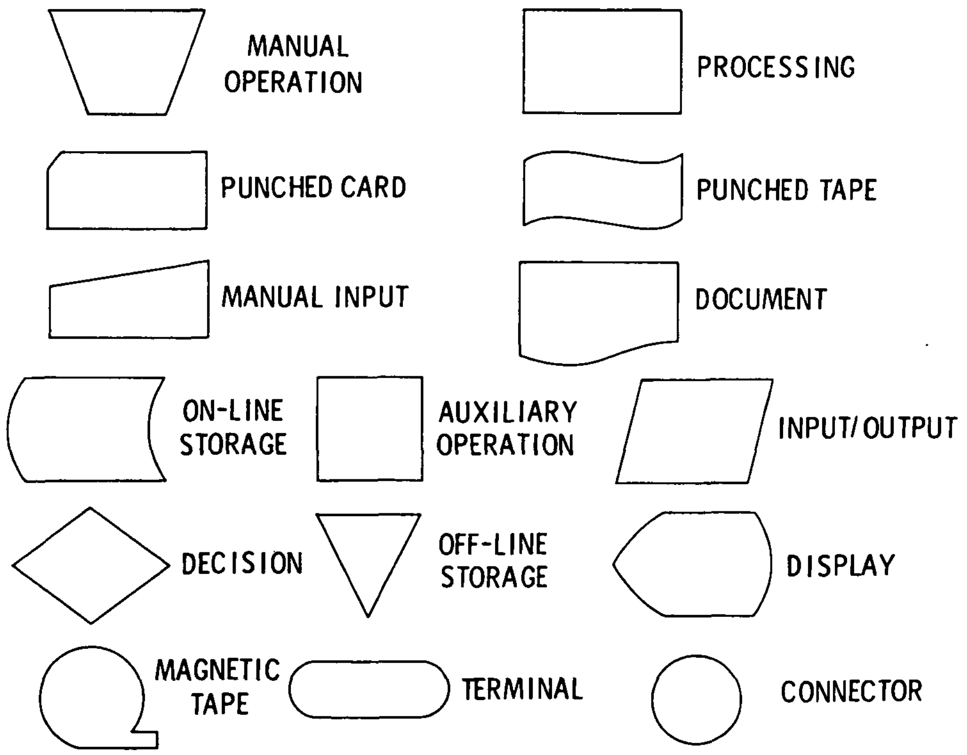


Figure 1.–Standard BELLFLOW symbols.

PRECEDING PAGE BLANK NOT FILMED

ADDITION OF NEW LANGUAGES

Another important reason for the development of BELLFLOW was the need to provide a flowcharting system structure that would allow the addition of new or unique special languages. There are many people at Bell Laboratories developing one-time languages or special languages that are unique to a very small area of design. These languages should be able to be conveniently added to the BELLFLOW system. By making BELLFLOW a model or table-driven system, a language has been developed that can be used for defining new programming languages in terms of BELLFLOW. As an example, consider the standard IF statement:

```
IF A = B THEN C = D
```

In BELLFLOW language, this would be

```
.B.IF.B..T..B.THEN.B..S.   /S(B*)
```

The following shows the entire set of model language statements required to define a programming language that is very close to PL/I structurally but not quite as complicated:

```
   .STATEMENT(";").
   .LABEL(":").
   .COMMENT("/*").
   .COMMENT(,"*/").
   (.B.IF.B..T..B.THEN.B..S.)   /S(*B)
   (B.ELSE.B..S)                /S(*L)
   (.B.GObTO.B..J.)             /S(*Z)
   (.B.DO.E.=.E.)               /S(*D)
   (.B.DO.B.)                   /S(*G)
   (.B.END.B.)                  /F(F1)
   (.B.END.B..J.)               /(*T)
F1 (.B.CALL.B)                  /S(*C)
   (.B.RETURN.B.)               /S(*Y)
   (.B.EXIT.B.)                 /S(*S)F(*X)
```

This set of statements is not supposed to provide an understanding of the details of the language itself but to show how straightforward it is to define a new programming language for the BELLFLOW system. For example, a new language called CENTRAN was recently added to BELLFLOW. It took about a week to develop the models, and, within another week or two, the system was essentially debugged and ready for use by programmers writing in the CENTRAN language.

USE OF GRAFPAC

Bell Laboratories uses a software package called GRAFPAC, which interfaces with all of our graphical devices. At the present time, the GRAFPAC system supports such devices as the SD 4060 and FR 80 microfilm plotters; the CALCOMP 718 and 728 and EAI 3500 line plotters; the Graphic 101 CRT terminal: STARE, an on-line hard-copy graphic unit that possesses quick turnaround features; and several others. The flowcharting system is interfaced through GRAFPAC so that, as new graphical devices are brought into Bell Laboratories and added to the GRAFPAC system, they are automatically available to the BELLFLOW system.

MULTIPLE-LEVEL FLOWCHARTS

Multiple-level flowcharts (very-high-system-level and very-low-detail-level) should also be imbeddable in the same source program. In addition, it is often desirable to imbed a separate flowchart for each procedure of a program even though all the procedures are compiled as a unit.

MODES OF OPERATIONS

There are three modes of operation: source mode, comment mode, and mixed mode. In the source mode, all of the flowcharting information is derived directly from the source code. This is similar to the way many other flowcharting systems operate. There is no reliance on comments or other information in the derivation of the entire flowchart.

In the comment mode, BELLFLOW ignores the source code completely and derives the entire flowchart purely from comments imbedded in the program. This has the advantage of being able to imbed comments in any program, whether or not the source language is supported by BELLFLOW. In the mixed mode, the source and comment mode are combined. The text that is to appear within a symbol and the definition of the symbol to be used are derived from a comment imbedded in the source code, but BELLFLOW uses the source code itself to determine the connections among the symbols and the placement and layout of the flowchart.

SELF-DOCUMENTING SOURCE DECK

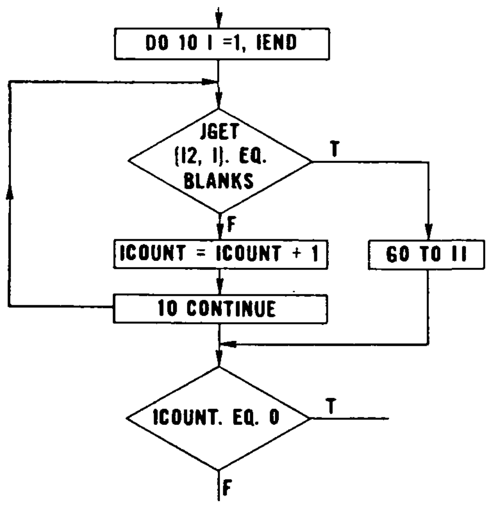
Because the mixed mode is unique to BELLFLOW, it may be of interest to explore the particular advantages of this mode. The goal was to try to provide a self-documenting program source deck. To achieve self-documentation, the flowchart should be imbedded in the source deck, and the source deck must be well commented. The question of what constitutes a "well-commented" program is too difficult to bring up here. In any event, most programmers use an intuitive definition of the term in deciding whether a program is well commented. A brief program description should also be imbedded at the beginning of the source deck. Given these three features, a subroutine or a program would contain most of the information needed for program documentation.

If the program flowchart is based on the source code alone (using the source mode), a very good two-dimensional representation and a poor functional description of the program are obtained. Many people feel that FORTRAN and other languages, such as PL/I, can be documented very nicely by placing the source code within the symbols, but this approach does not seem very fruitful.

If the flowchart is based only on comments that are imbedded in the program (using the comment mode), the result may be an excellent functional description of the program

```
      DO 10 I = 1, IEND

      IF (JGET (12, I) .EQ. BLANKS) GO TO 11

      ICOUNT = ICOUNT + 1

10    CONTINUE

11    IF (ICOUNT .EQ. 0) ICOUNT = 1
```

(a)



(b)

Figure 2.–(a) FORTRAN source mode. (b) Corresponding flowchart.

in understandable language, but no real tie-in between those comments and the program itself. It is not certain that the comments reflect the flow and interconnectivity of the program. Furthermore, the programmers usually have to supply a lot of redundant information in the comments to express the interconnectivity among the symbols.

The mixed mode permits one to put text inside symbols in a way that is meaningful to a person trying to understand a program, and yet, the fact that the program flow (the interconnectivity among symbols) is derived from the source code itself is still retained. No other comments other than the BELLFLOW comments are required to produce a well-commented program. If a preface is added at the beginning of the program (a manual operation), a self-documenting source program deck is obtained.

EXAMPLES OF BELLFLOW

Figure 2(a) is an example of the source mode in FORTRAN. Figure 2(b) shows the corresponding flowchart that would be generated. Quite often, the source mode is used when improvements have to be made in an old FORTRAN program. Because it is not always possible to locate the original program writer, it is very convenient to be able to obtain a two-dimensional flowchart listing in the source mode; this makes the task much easier than it would be if a linear listing of the program had to be used.

The comment mode format is

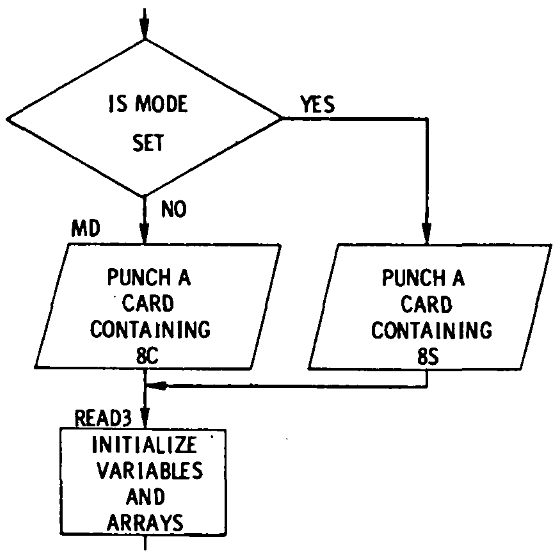
```
*F(LEVEL)(LABEL) TEXT /SYMBOL/OUTPUT
```

The comment begins with an indicator that is the normal comment indicator of the programming language. For an assembly language, the indicator might be an asterisk in the first column. The letter F indicates that a flowchart comment is being made. The next field is a level indicator that allows one to imbed more than one set of flowchart comments; it is a two-character identifier that denotes a particular flowchart. If there is only one flowchart imbedded in the program, this field may be left blank. The label field is usually left blank unless it is needed to specify interconnectivity.

The main body in this statement is the comment text field, which conveys the information that is to appear in the symbol. The final two fields at the end of the statement are the symbol field and the output field. The output field contains information that is to be placed

```
*F          IS MODE SET                    /D/NO(MD)YES
*F          PUNCH A CARD CONTAINING 8S     /IO/(READ3)
*F   MD     PUNCH A CARD CONTAINING 8C     /IO
*F   READ3  INITIALIZE VARIABLES AND ARRAYS
```

(a)


IS MODE SET
YES
NO
MD
PUNCH A CARD CONTAINING 8C
PUNCH A CARD CONTAINING 8S
READ3
INITIALIZE VARIABLES AND ARRAYS

(b)

Figure 3.–(a) Imbedded comments. (b) Typical comment mode flowchart.

on output branches emanating from symbols, and, in some cases, it specifies the interconnectivity between one symbol and another.

An example of the comment mode format is

```
* F IS COUNT=MAX+TOL  /D/YES,NO
```

Here, the level is blank, and there is no label. The letter D in the symbol field indicates a decision symbol. One branch will have the label YES, and the other branch will have the label NO. Figure 3 shows a typical portion of a comment mode flowchart.

The comment mode is extremely useful in preliminary design for the generation of system flowcharts when no code exists or as part of the documentation of an early design. A complicated flowchart can be quickly encoded, run off, modified, edited over a period of meetings, and kept up to date by using the comment mode without ever having any source code. The comment mode is also being used to generate nonprogramming documentation such as flowchart-like and program evaluation and review technique drawings, and input/output relation tables.

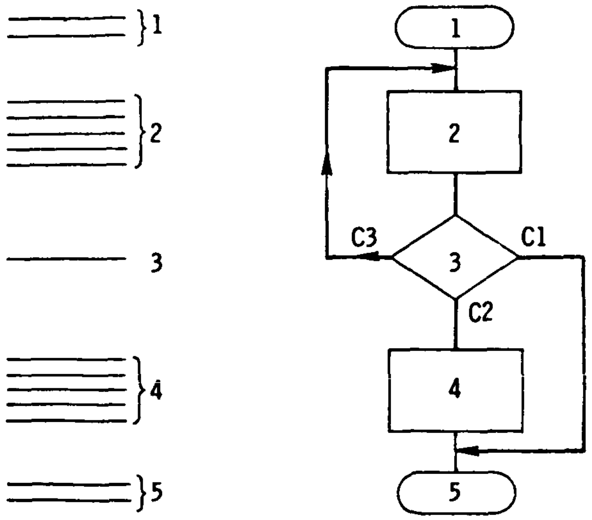


Figure 4.–Correspondence of program to flowchart.

Figure 4 is a flowchart consisting of five symbols. The lines on the left are used to indicate statements of source code and are grouped to indicate which statements correspond to the five symbols in the flowchart. In the mixed mode, a BELLFLOW comment is inserted at the beginning of each one of these functional groupings of source code.

Figure 5(a) is an example of the mixed mode. Note that the comments are very similar to the ones that might be placed in a program even if BELLFLOW were not being used. The second BELLFLOW comment, "ARE SIGNS EQUAL," is the text for a decision symbol with two branches labeled YES and NO. There are two ways to exit from this functional block. The first one is encountered at the skip instruction (SPA), and the second one is encountered at the jump (JMP), so the YES text is associated with the first way to exit and the NO text is associated with the second way to exit. Figure 5(b) shows the corresponding flowchart. It is important to repeat at this point that the BELLFLOW comments are really all the comments that are needed to provide a well-commented source deck.

BELLFLOW FEATURES

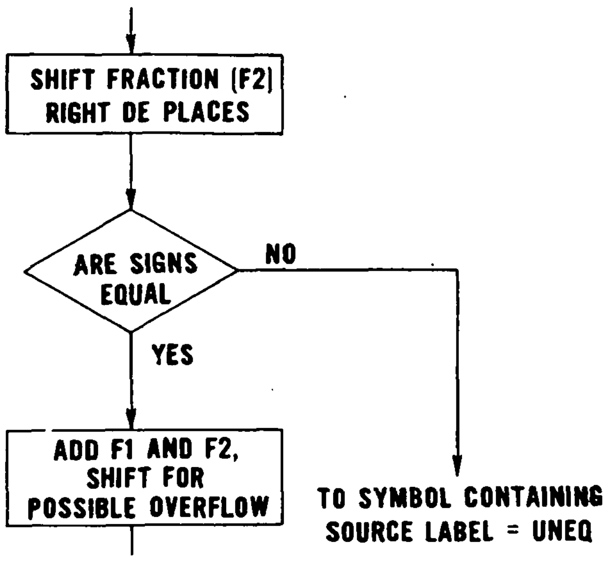
In addition to automatic placement, automatic line routing, paging, and generation of on- and off-sheet connectors, all of which are standard regardless of the mode employed, BELLFLOW possesses some additional features. One can request left-to-right rather than top-to-bottom flow in the flowchart. Normally, BELLFLOW will automatically format the text that goes in a symbol, but the programmer can do this himself if he considers it desirable.

BELLFLOW will adhere to standard aspect ratios in choosing symbol sizes but also has the capability of using nonstandard symbol sizes. In the latter case, processing symbols (rectangles, for example), are reduced to fit exactly around the text provided. It is interesting to note that, in a brief study of this capability, a gain of almost 50 percent in the number of symbols per vertical row was achieved with the use of the nonstandard symbol sizes. Increasing the number of symbols per sheet reduces the number of off-sheet connectors and, therefore, simplifies the flow and makes it more readable.

BELLFLOW permits multiple keypunch codes to be used, and an option is provided that permits flowcharts to be made without any crossovers. The normal mode will make crossovers unless too many lines are crossed, in which case an on-sheet connector is generated.

```
        DAC      F2
*
*F               SHIFT FRACTION (F2) RIGHT DE PLACES
*
SHIFT   LAC      DE
        TAD      SHIFT1
        DAC      *+2
        LAC      F2
        CSS      1
        DAC      F2
*
*F               ARE SIGNS EQUAL                    /D/YES,NO
*
        LAC      F1
        XOR      F2
        SPA
        JMP      UNEQ
*
*F               ADD F1 AND F2,SHIFT FOR POSSIBLE OVERFLOW
*
        CLL
         .
         .
```

(a)


SHIFT FRACTION (F2) RIGHT DE PLACES
ARE SIGNS EQUAL
NO
YES
ADD F1 AND F2, SHIFT FOR POSSIBLE OVERFLOW
TO SYMBOL CONTAINING SOURCE LABEL = UNEQ

(b)

Figure 5.–(a) Comments. (b) Flowchart for BELLFLOW mixed mode example.

SAMPLE OUTPUTS

Figure 6 is an example of the BELLFLOW output produced by the FR 80 microfilm plotter. Note the placement, line routing, and general quality features of the flowchart. The cost of such a flowchart sheet, including the analysis and drawing, is about $2 to $3 when the IBM 360/85 is used.

Figure 7 is another example of the BELLFLOW output; this one was made by a CALCOMP line plotter. It is larger than the sheet shown in Figure 6 and has about 50 symbols. Again, the important things to note are the placement, line routing, and general quality features of the flowchart. Figure 8 shows the same flowchart that Figure 7 does, but it was made with left-to-right rather than vertical flow.

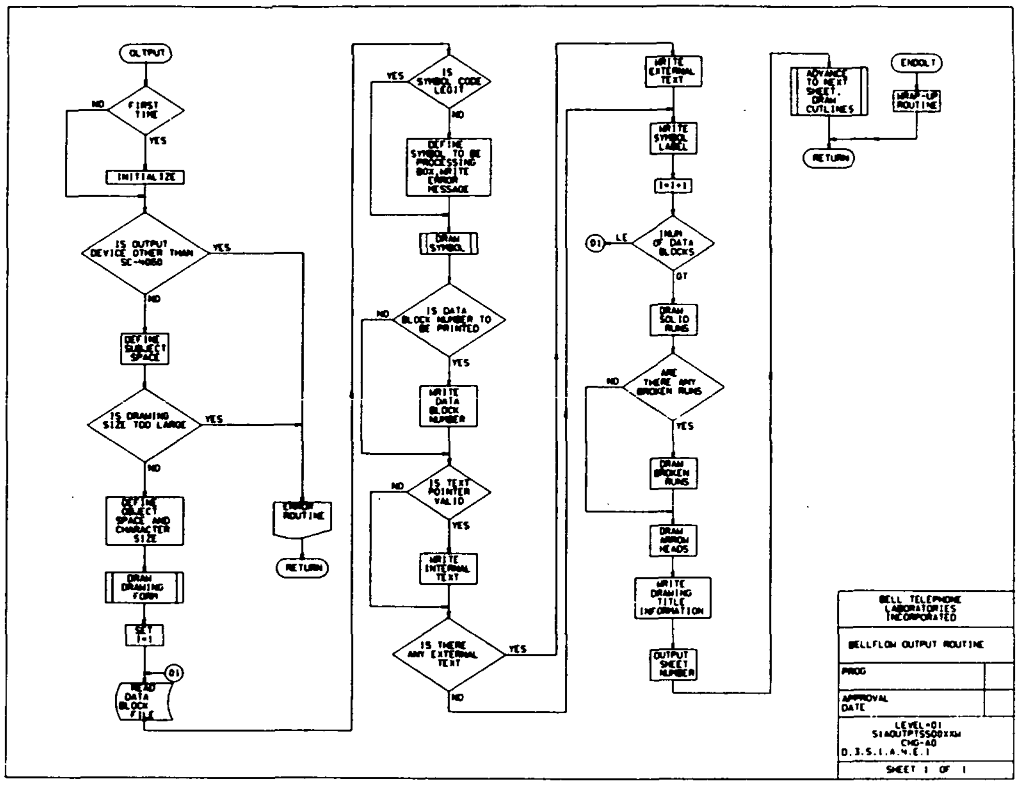


Figure 6.—Sample FR 80 microfilm output.

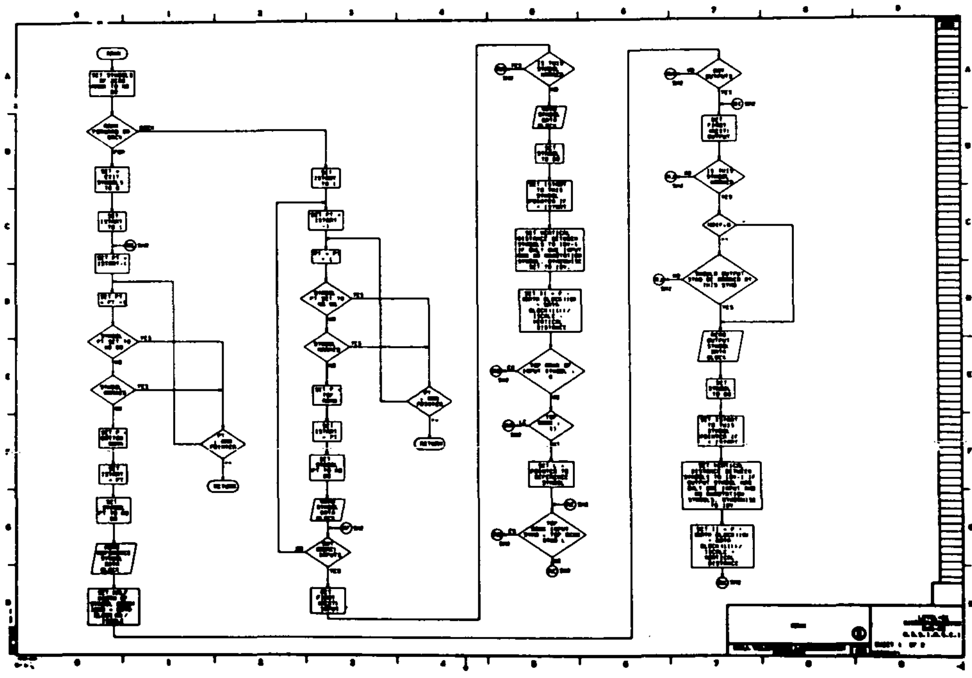


Figure 7.—Sample CALCOMP output.

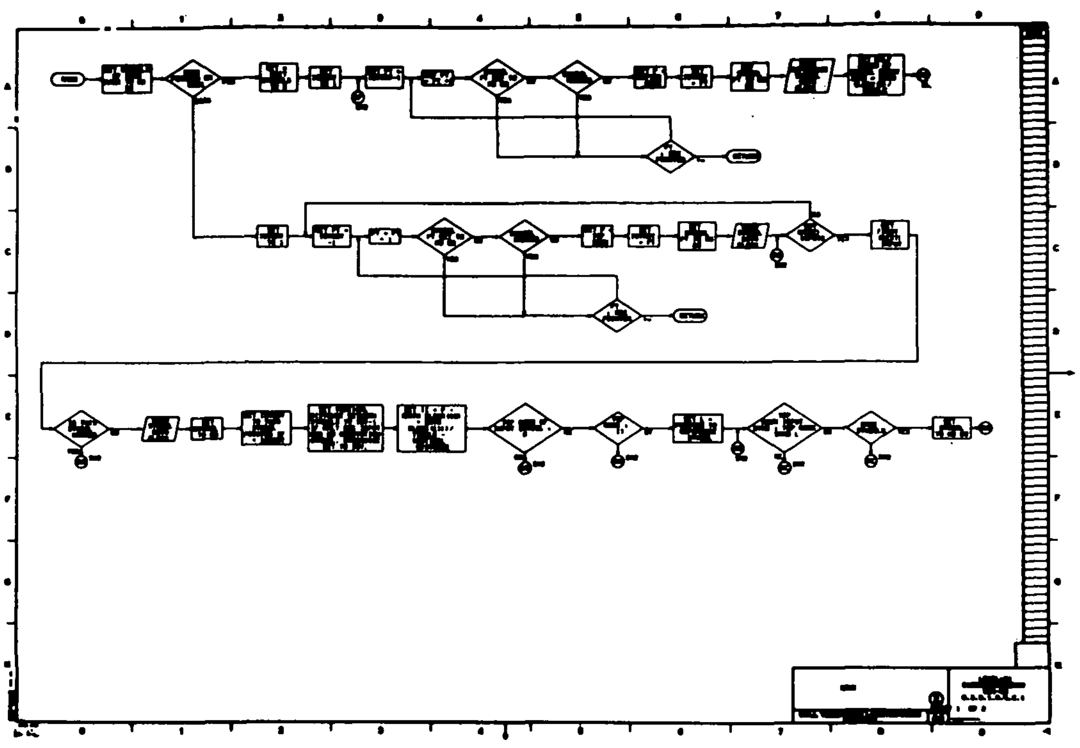


Figure 8.–Sample CALCOMP output, left-to-right flow.

CONCLUSION

The BELLFLOW system has been working for about a year and a half. One project manager, who heads a rather sizable software development project, stated that with BELLFLOW he felt that he had an adequate means for insuring that he had both a well-commented and properly flowcharted program. He insists that programmers must deliver a flowchart, in the mixed mode, to him before he will accept their program and consider it complete. By following this procedure, he reasons that, if a good flowchart can be produced, then the comments that appear in the deck are also understandable, meaningful, and useful. So he accomplishes the dual goal of having a program well-commented and properly flowcharted by using BELLFLOW as an administrative tool.

DISCUSSION

MEMBER OF THE AUDIENCE: In the event that the comments and the actual operation statements do not match, is there an editing function in the BELLFLOW system that would point this out?

PARDEE: There is no specific function that points this out.

MEMBER OF THE AUDIENCE: I would like to ask the same question I asked Goetz: Do you document BELLFLOW with BELLFLOW?

PARDEE: Yes.

MEMBER OF THE AUDIENCE: If the syntax of the modal language is unambiguous in the program mode, would it be possible to program from the flowcharts using the logic to work back to a program? Also, is it possible to alter the placement of things on flowcharts?

PARDEE: Programming from the flowchart is something that we are going to study. We do not know enough yet to be able to say whether it might be helpful to input a flowchart and then have the machine automatically convert the logic into a source code. There is a manual placement option that can exert a certain amount of control in the relative placement of symbols, but it does not control specific placement. We are investigating the possibility of a manual touchup capability that would be able to make specific changes in the flowchart.

MEMBER OF THE AUDIENCE: Why was not any mention made of the IBM 360/67?

PARDEE: Our philosophy was to detach ourselves from the compiler and assembler functions to remain somewhat independent of the machines and any changes involving compilers and assemblers and to deal only basically with the language, which was fairly well defined.

MEMBER OF THE AUDIENCE: Do you have any plans for making BELLFLOW a commercial product?

PARDEE: No. There is the possibility of people having access to BELLFLOW, however.

N73-19208

AN AUTOMATED SYSTEM FOR GENERATING PROGRAM DOCUMENTATION

Richard J. Hanney
Grumman Data Systems

There are several proprietary software packages available that provide the user with documentation aids. IBM's AUTOCHART and Applied Data Research's AUTOFLOW are examples of flowchart generators. For many years, it has been the flowchart that provides the key to good program documentation. Flowcharts are invaluable for documenting machine language programs; they are only slightly less helpful with large FORTRAN and COBOL programs. However, flowcharts are not necessarily the most important pieces of documentation for medium-size, compiler-level programs.

With this in mind, a documentation program was developed in which the emphasis is placed on text content rather than flowcharting. Grumman Data Systems has been using this program for 1 year to document most of its production-type programs. There are personnel whose sole responsibility is to prepare these production jobs for computer runs. Each of these individuals must know how to prepare several different jobs for runs on the computer. This arrangement permits the programmer to write, debug, and then turn over his program along with a copy of his documentation to deck assembly personnel for production use.

The documentation that accompanies the debugged program is often called operational documentation. Such documentation usually includes all parts of the complete document except the source listings. A typical document would include the following:

(1) Accounting information (deck number, job charge numbers, etc.)
(2) General description (an abstract of the program explaining how it fits into the overall data flow and subroutine descriptions)
(3) Functional flowcharts (Detailed flowcharts are not considered necessary because of the type of work done by batch programmers. Few programs of any size are reused once their primary job requirement has ended. In most cases, if a future task has need for a similar program, a new program is written, rather than rework an old program.)
(4) Data card formats, deck setups, and options

These four sections appear in all operational documentation. The programming department retains a copy of the document with the source listings added.

USE OF THE DOCUMENTATION PROGRAM

The documentation program is used to generate the entire document. It is keyword oriented, with 26 keywords that control the program. Seventeen of those keywords are recognized by the flowchart generator, three are related to text generation, and three have to do with control card and deck displays. The programmer who uses the documentation program prepares his document on data cards. This is the major drawback of the documentation program because all input is contained on 80-column card images that must be prepared by the programmer. The program cannot yet generate text narratives by inspecting source decks.

Repetitive use of the documentation program helps each programmer become more familiar with both this particular program and program documentation in general. The keyword cards are very easy to understand and prepare, and most programmers become adept at using the program after their first try. The strongest advantage offered by the documentation program is that it produces the entire document. The document is prepared on 35-mm microfilm, which is easy to store, and letter-size reproductions can be made inexpensively on bond paper. The following is a list of features of the documentation program:

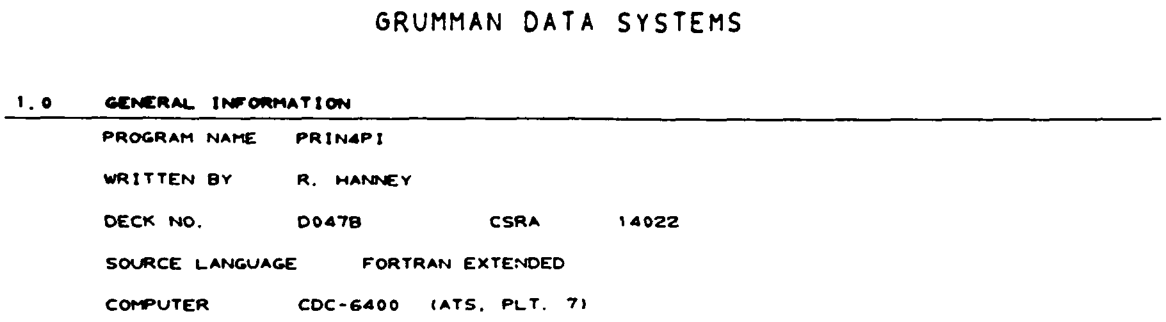
(1) *Text generator*—the program can delimit sections of a document at three levels: topics, subtopics, and paragraphs.
(2) *Flowchart generator*—the flowchart generator is activated when the program reads a flow card (one of the 17 flowchart keywords). Subsequent cards are inspected for special keywords that pertain to flowcharting. Flowcharting terminates when an exit card is read. The flowchart generator can be called as many times as desired.
(3) *Coding form displays*—cards can be displayed singly or in groups on a coding form display which numbers each column.
(4) *Deck displays*—the deck option operates exactly like the card option, with one added feature: After the coding form display has been completed, the deck option will cause all displayed cards to be shown in a fanned-deck pictorial representation. This type of display gives quick information about card orders to personnel responsible for job preparation.
(5) *Underlining*—any line of text can be underlined for particular emphasis of important words or phrases.
(6) *Sample output display*—the programmer can include a binary coded decimal file containing sample tab outputs of his program. The documentation program will automatically include the samples in the document when it reads an output card.
(7) *Auxiliary tape input*—a keyword card will cause the program to switch from the standard input file (usually the card reader) to any other sequential file for primary input card images.
(8) *Index*—an index of all topics, subtopics, and paragraphs is provided at the end of the document that gives page numbers of all sections of the document.

To aid the programmer in checking the output of the documentation program, a tab listing is provided that gives the results of the run. Usually, the microfilm output file is stored on tape; the programmer can direct this file to the microfilm unit after he has checked the tap output and is satisfied that his document is correct. The tab listing shows the contents of each frame of output generated by the documentation program. Flowcharts are also printed to show exactly how they will appear on the film. Figure 1 contains a sample program output.

FUTURE DEVELOPMENTS

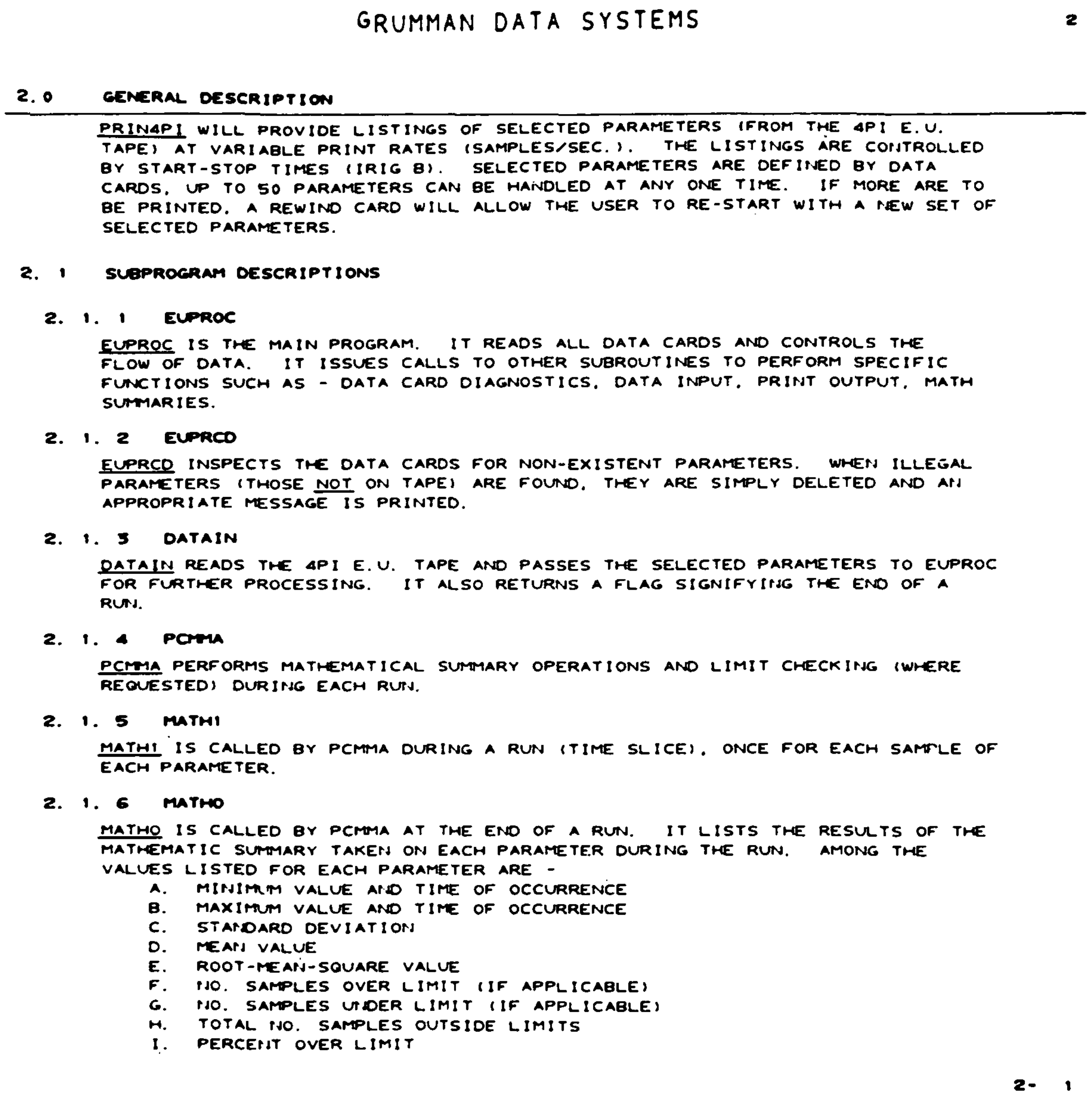
The features of the described documentation program are options; for example, the user does not have to use the flowchart generator. Thus, program documentation is still controlled by the programmer. To prevent too many stylized documents from being generated, the documentation program is supplemented by a set of documentation standards to which all programmers must adhere. Because the documentation program itself is in total control of the various formats of its options, program documentation has become fairly standardized; at least the formats are similar, although content and quality are still the responsibility of the programmer.

It is clear that this system uses a traditional rather than a new or radical approach to the problem of documentation. However, these methods suit Grumman's internal needs.



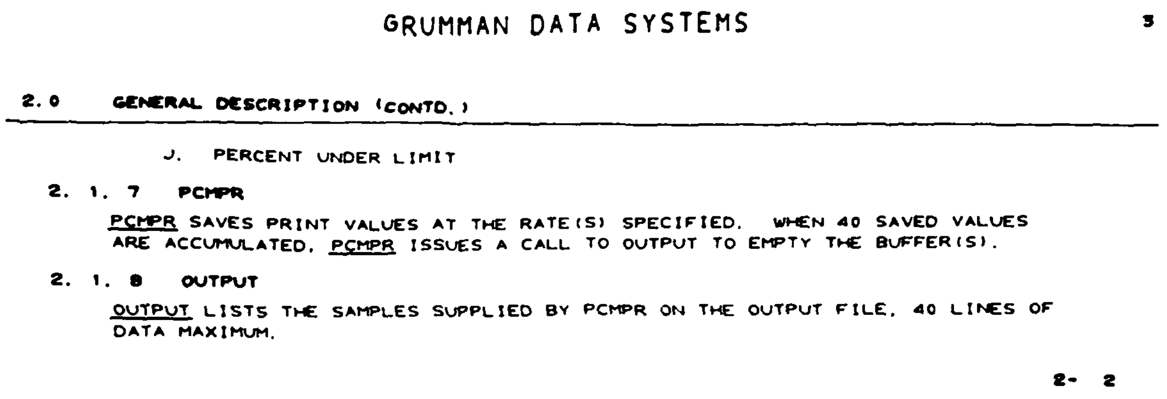
```
                         GRUMMAN DATA SYSTEMS

1.0    GENERAL INFORMATION
------------------------------------------------------------------------
       PROGRAM NAME    PRIN4PI

       WRITTEN BY      R. HANNEY

       DECK NO.        D047B          CSRA        14022

       SOURCE LANGUAGE      FORTRAN EXTENDED

       COMPUTER        CDC-6400   (ATS. PLT. 7)
```

Figure 1.–Illustration of documentation program.



```
                              GRUMMAN DATA SYSTEMS                              2

2.0    GENERAL DESCRIPTION
       PRIN4PI WILL PROVIDE LISTINGS OF SELECTED PARAMETERS (FROM THE 4PI E.U.
       TAPE) AT VARIABLE PRINT RATES (SAMPLES/SEC.).  THE LISTINGS ARE CONTROLLED
       BY START-STOP TIMES (IRIG B).  SELECTED PARAMETERS ARE DEFINED BY DATA
       CARDS, UP TO 50 PARAMETERS CAN BE HANDLED AT ANY ONE TIME.  IF MORE ARE TO
       BE PRINTED, A REWIND CARD WILL ALLOW THE USER TO RE-START WITH A NEW SET OF
       SELECTED PARAMETERS.

2. 1   SUBPROGRAM DESCRIPTIONS

  2. 1. 1   EUPROC
       EUPROC IS THE MAIN PROGRAM.  IT READS ALL DATA CARDS AND CONTROLS THE
       FLOW OF DATA.  IT ISSUES CALLS TO OTHER SUBROUTINES TO PERFORM SPECIFIC
       FUNCTIONS SUCH AS - DATA CARD DIAGNOSTICS, DATA INPUT, PRINT OUTPUT, MATH
       SUMMARIES.

  2. 1. 2   EUPRCD
       EUPRCD INSPECTS THE DATA CARDS FOR NON-EXISTENT PARAMETERS.  WHEN ILLEGAL
       PARAMETERS (THOSE NOT ON TAPE) ARE FOUND, THEY ARE SIMPLY DELETED AND AN
       APPROPRIATE MESSAGE IS PRINTED.

  2. 1. 3   DATAIN
       DATAIN READS THE 4PI E.U. TAPE AND PASSES THE SELECTED PARAMETERS TO EUPROC
       FOR FURTHER PROCESSING.  IT ALSO RETURNS A FLAG SIGNIFYING THE END OF A
       RUN.

  2. 1. 4   PCMMA
       PCMMA PERFORMS MATHEMATICAL SUMMARY OPERATIONS AND LIMIT CHECKING (WHERE
       REQUESTED) DURING EACH RUN.

  2. 1. 5   MATH1
       MATH1 IS CALLED BY PCMMA DURING A RUN (TIME SLICE), ONCE FOR EACH SAMPLE OF
       EACH PARAMETER.

  2. 1. 6   MATHO
       MATHO IS CALLED BY PCMMA AT THE END OF A RUN.  IT LISTS THE RESULTS OF THE
       MATHEMATIC SUMMARY TAKEN ON EACH PARAMETER DURING THE RUN.  AMONG THE
       VALUES LISTED FOR EACH PARAMETER ARE -
            A.   MINIMUM VALUE AND TIME OF OCCURRENCE
            B.   MAXIMUM VALUE AND TIME OF OCCURRENCE
            C.   STANDARD DEVIATION
            D.   MEAN VALUE
            E.   ROOT-MEAN-SQUARE VALUE
            F.   NO. SAMPLES OVER LIMIT (IF APPLICABLE)
            G.   NO. SAMPLES UNDER LIMIT (IF APPLICABLE)
            H.   TOTAL NO. SAMPLES OUTSIDE LIMITS
            I.   PERCENT OVER LIMIT

                                                                            2-  1
```

Figure 1 (continued).–Illustration of documentation program.



```
                         GRUMMAN DATA SYSTEMS                          3

2.0     GENERAL DESCRIPTION (CONTD.)
_______________________________________________________________________
                J.  PERCENT UNDER LIMIT

  2. 1. 7   PCMPR

        PCMPR SAVES PRINT VALUES AT THE RATE(S) SPECIFIED.  WHEN 40 SAVED VALUES
        ARE ACCUMULATED, PCMPR ISSUES A CALL TO OUTPUT TO EMPTY THE BUFFER(S).

  2. 1. 8   OUTPUT

        OUTPUT LISTS THE SAMPLES SUPPLIED BY PCMPR ON THE OUTPUT FILE, 40 LINES OF
        DATA MAXIMUM.

                                                                     2- 2
```

Figure 1 (continued).–Illustration of documentation program.

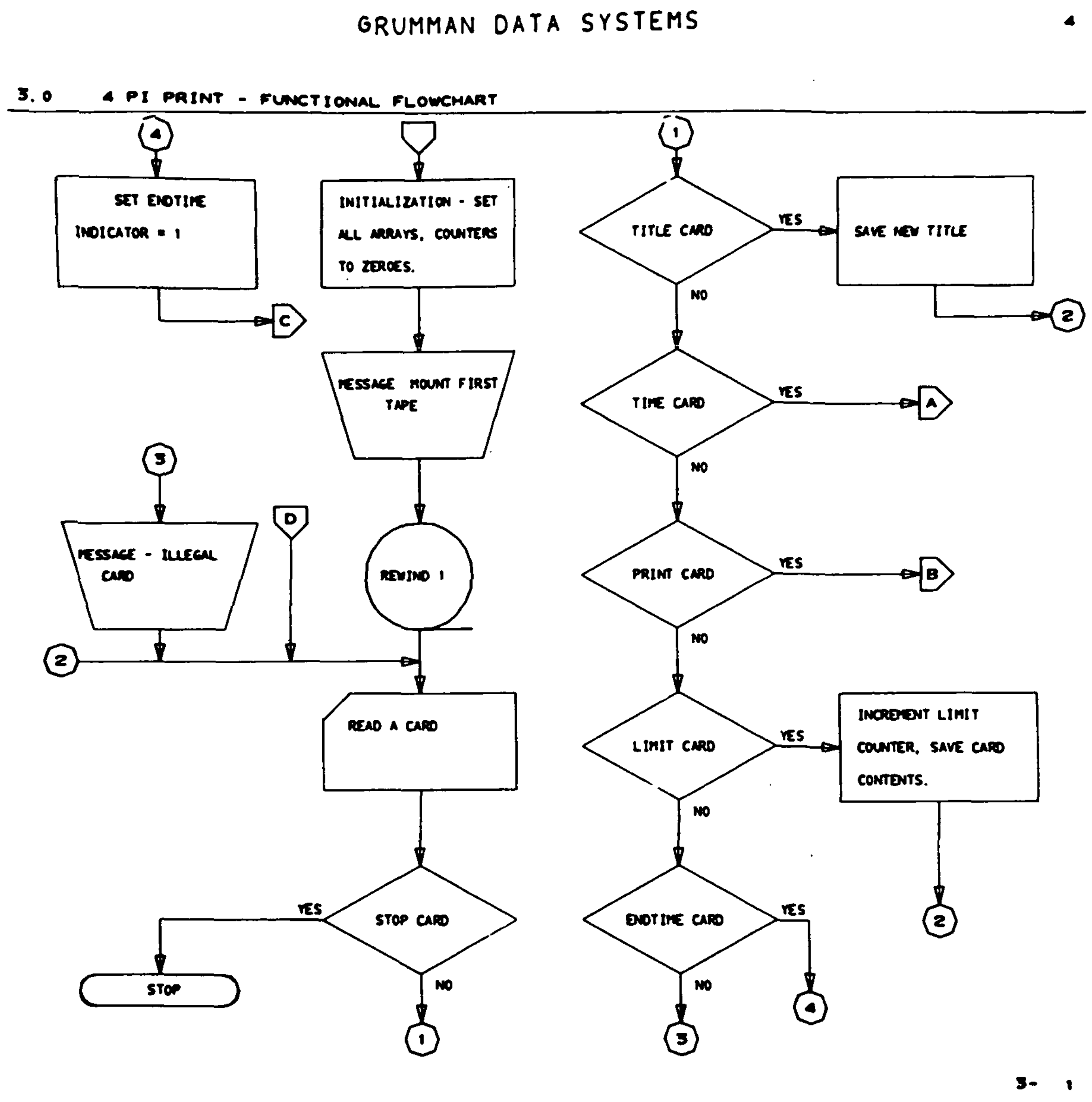


Figure 1 (continued).–Illustration of documentation program.

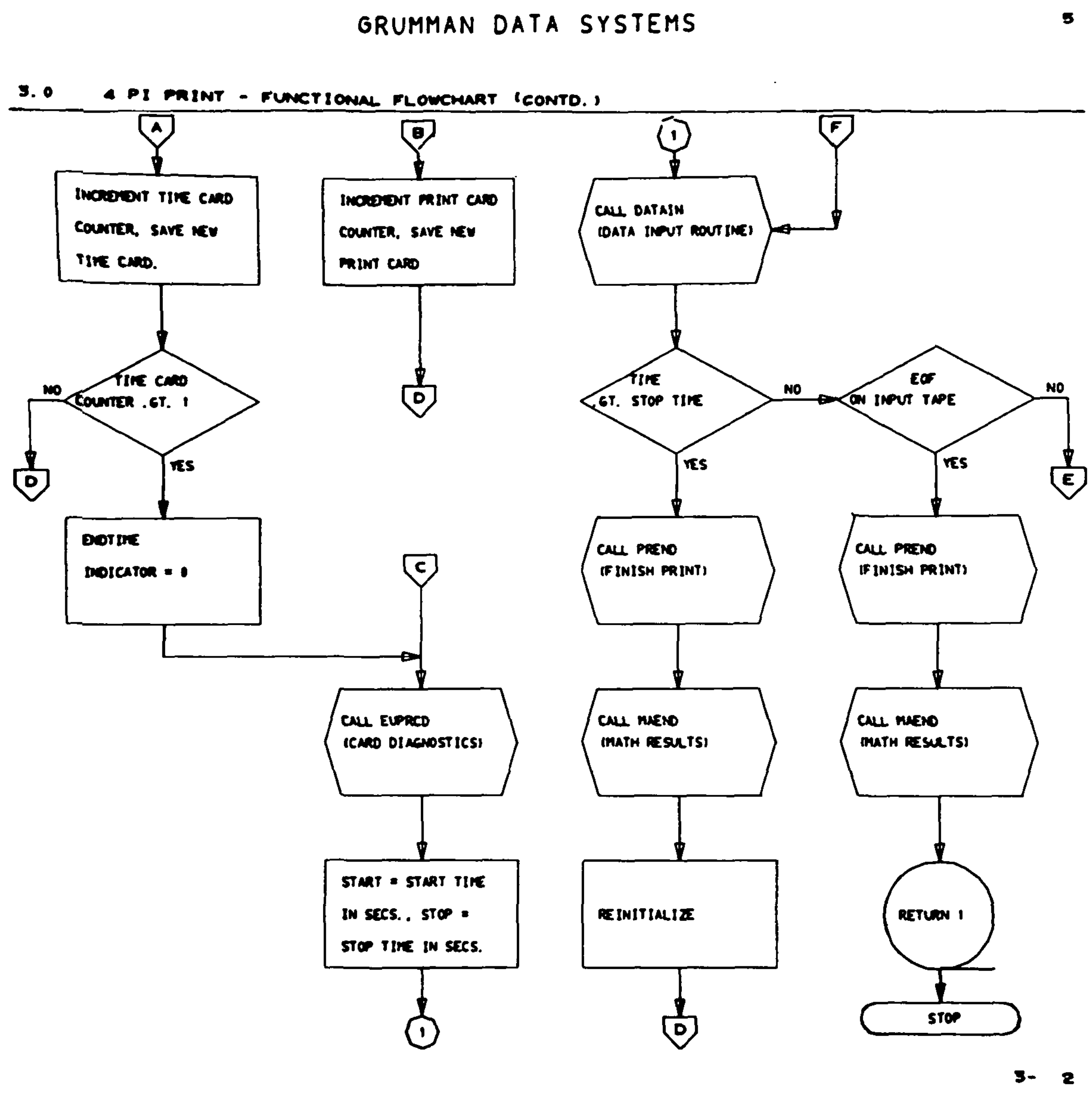
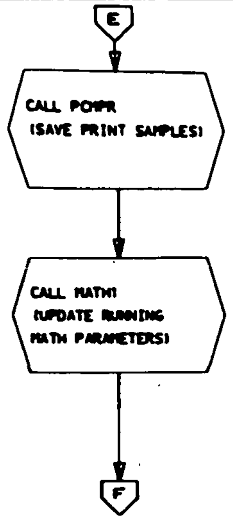

GRUMMAN DATA SYSTEMS
5
3.0 4 PI PRINT - FUNCTIONAL FLOWCHART (CONTD.)
A
INCREMENT TIME CARD COUNTER, SAVE NEW TIME CARD.
TIME CARD COUNTER .GT. 1
NO
YES
D
ENDTIME INDICATOR = 0
B
INCREMENT PRINT CARD COUNTER, SAVE NEW PRINT CARD
D
C
CALL EUPRCD (CARD DIAGNOSTICS)
START = START TIME IN SECS., STOP = STOP TIME IN SECS.
1
1
F
CALL DATAIN (DATA INPUT ROUTINE)
TIME .GT. STOP TIME
NO
YES
CALL PREND (FINISH PRINT)
CALL MAEND (MATH RESULTS)
REINITIALIZE
D
EOF ON INPUT TAPE
NO
YES
E
CALL PREND (FINISH PRINT)
CALL MAEND (MATH RESULTS)
RETURN 1
STOP
3- 2

Figure 1 (continued).–Illustration of documentation program.

3.0 4 PI PRINT - FUNCTIONAL FLOWCHART (CONTD.)



3- 3

Figure 1 (continued).–Illustration of documentation program.

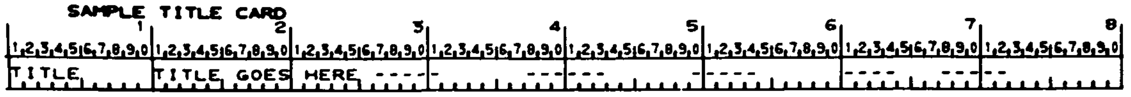
4.0 CARD FORMATS

THE CARDS DISPLAYED IN THIS SECTION ARE RECOGNIZED BY THE PROGRAM. ALL OTHER CARDS WILL BE REJECTED.

4. 1 TITLE CARD

THE TITLE CARD SHOULD BE THE FIRST CARD IN THE DECK. IT CONTAINS A TITLE WHICH WILL APPEAR AT THE TOP OF EACH PAGE OF OUTPUT.

SAMPLE TITLE CARD

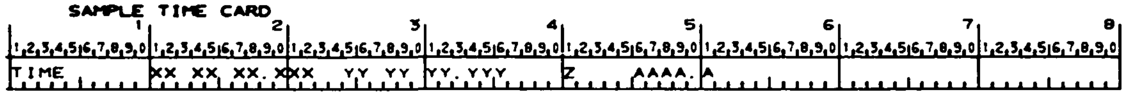


COLS.	CONTENTS
1-5	TITLE
11-72	TITLE TO APPEAR AT TOP OF EACH PAGE OF OUTPUT

4. 2 TIME SLICE CARD

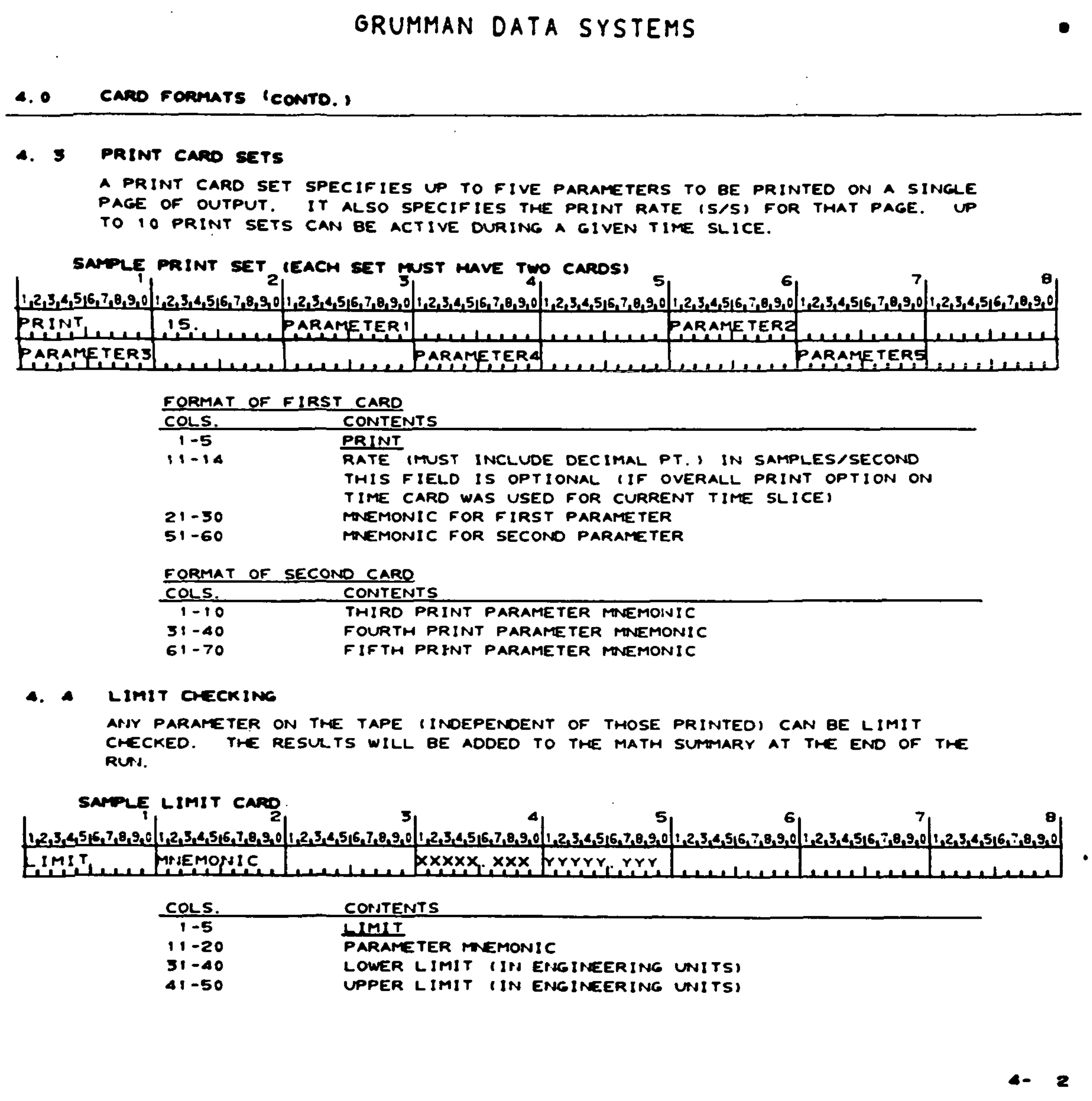
THE TIME CARD DEFINES THE IRIG START-STOP TIMES FOR A RUN. IT ALSO ALLOWS THE USER TO INHIBIT THE MATH SUMMARY WHICH IS COMPILED DURING THE RUN, AND (IF DESIRED) TO SPECIFY ONE PRINT RATE FOR ALL PRINT PARAMETERS.

SAMPLE TIME CARD



COLS.		CONTENTS
1-4		TIME
11-12	---	
14-15	----	START TIME IN HOURS (11-12), MIN (14-15), SECS (17-22)
17-22	---	
25-26	---	
28-29	----	STOP TIME IN HOURS (25-26), MIN (28-29), SECS (31-36)
31-36	---	
41		MATH INHIBIT FLAG = 1, NO MATH OUTPUT AT END OF SLICE
46-51		OVERALL PRINT RATE (MUST INCLUDE DEC. PT.) IN SAMPLES/SECOND

Figure 1 (continued).–Illustration of documentation program.



GRUMMAN DATA SYSTEMS

4.0 CARD FORMATS (CONTD.)

4.3 PRINT CARD SETS

A PRINT CARD SET SPECIFIES UP TO FIVE PARAMETERS TO BE PRINTED ON A SINGLE PAGE OF OUTPUT. IT ALSO SPECIFIES THE PRINT RATE (S/S) FOR THAT PAGE. UP TO 10 PRINT SETS CAN BE ACTIVE DURING A GIVEN TIME SLICE.

SAMPLE PRINT SET (EACH SET MUST HAVE TWO CARDS)

1,2,3,4,5,6,7,8,9,0 (1)	1,2,3,4,5,6,7,8,9,0 (2)	1,2,3,4,5,6,7,8,9,0 (3)	1,2,3,4,5,6,7,8,9,0 (4)	1,2,3,4,5,6,7,8,9,0 (5)	1,2,3,4,5,6,7,8,9,0 (6)	1,2,3,4,5,6,7,8,9,0 (7)	1,2,3,4,5,6,7,8,9,0 (8)
PRINT	15.	PARAMETER1			PARAMETER2		
PARAMETER3			PARAMETER4			PARAMETER5	

FORMAT OF FIRST CARD

COLS.	CONTENTS
1-5	PRINT
11-14	RATE (MUST INCLUDE DECIMAL PT.) IN SAMPLES/SECOND THIS FIELD IS OPTIONAL (IF OVERALL PRINT OPTION ON TIME CARD WAS USED FOR CURRENT TIME SLICE)
21-30	MNEMONIC FOR FIRST PARAMETER
51-60	MNEMONIC FOR SECOND PARAMETER

FORMAT OF SECOND CARD

COLS.	CONTENTS
1-10	THIRD PRINT PARAMETER MNEMONIC
31-40	FOURTH PRINT PARAMETER MNEMONIC
61-70	FIFTH PRINT PARAMETER MNEMONIC

4.4 LIMIT CHECKING

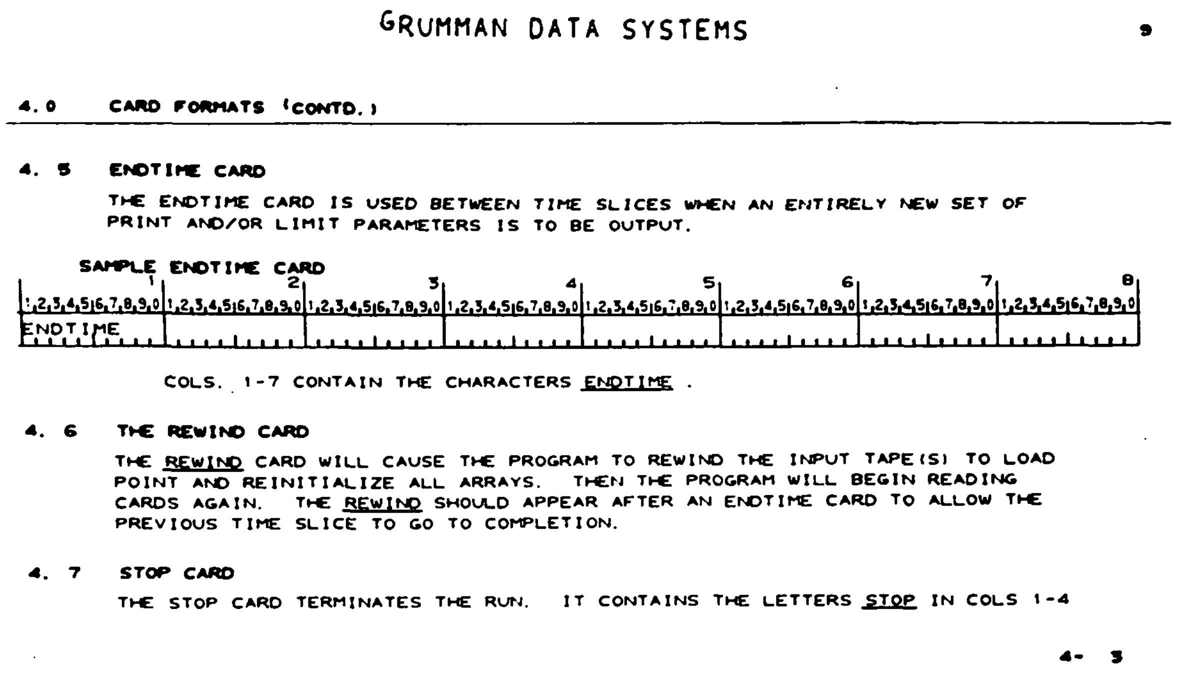
ANY PARAMETER ON THE TAPE (INDEPENDENT OF THOSE PRINTED) CAN BE LIMIT CHECKED. THE RESULTS WILL BE ADDED TO THE MATH SUMMARY AT THE END OF THE RUN.

SAMPLE LIMIT CARD

1,2,3,4,5,6,7,8,9,0 (1)	1,2,3,4,5,6,7,8,9,0 (2)	1,2,3,4,5,6,7,8,9,0 (3)	1,2,3,4,5,6,7,8,9,0 (4)	1,2,3,4,5,6,7,8,9,0 (5)	1,2,3,4,5,6,7,8,9,0 (6)	1,2,3,4,5,6,7,8,9,0 (7)	1,2,3,4,5,6,7,8,9,0 (8)
LIMIT	MNEMONIC		XXXXX.XXX	YYYYY.YYY			

COLS.	CONTENTS
1-5	LIMIT
11-20	PARAMETER MNEMONIC
31-40	LOWER LIMIT (IN ENGINEERING UNITS)
41-50	UPPER LIMIT (IN ENGINEERING UNITS)

Figure 1 (continued).—Illustration of documentation program.



GRUMMAN DATA SYSTEMS 9

4.0 CARD FORMATS (CONTD.)

4. 5 ENDTIME CARD

THE ENDTIME CARD IS USED BETWEEN TIME SLICES WHEN AN ENTIRELY NEW SET OF PRINT AND/OR LIMIT PARAMETERS IS TO BE OUTPUT.

SAMPLE ENDTIME CARD

```
         1         2         3         4         5         6         7         8
1234567890123456789012345678901234567890123456789012345678901234567890123456789 0
ENDTIME
```

COLS. 1-7 CONTAIN THE CHARACTERS ENDTIME .

4. 6 THE REWIND CARD

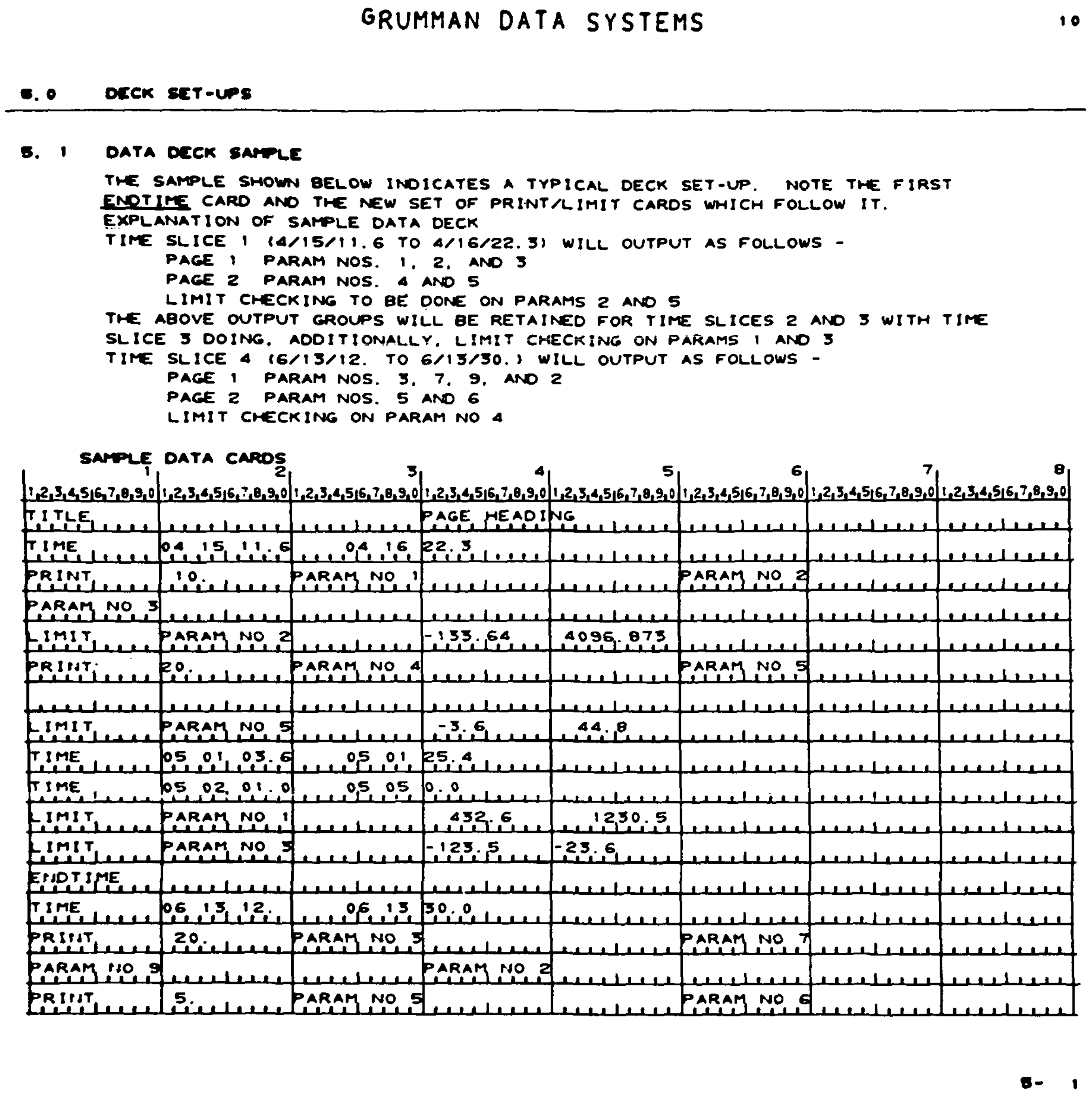
THE REWIND CARD WILL CAUSE THE PROGRAM TO REWIND THE INPUT TAPE(S) TO LOAD POINT AND REINITIALIZE ALL ARRAYS. THEN THE PROGRAM WILL BEGIN READING CARDS AGAIN. THE REWIND SHOULD APPEAR AFTER AN ENDTIME CARD TO ALLOW THE PREVIOUS TIME SLICE TO GO TO COMPLETION.

4. 7 STOP CARD

THE STOP CARD TERMINATES THE RUN. IT CONTAINS THE LETTERS STOP IN COLS 1-4

4- 5

Figure 1 (continued).–Illustration of documentation program.



5.0 DECK SET-UPS

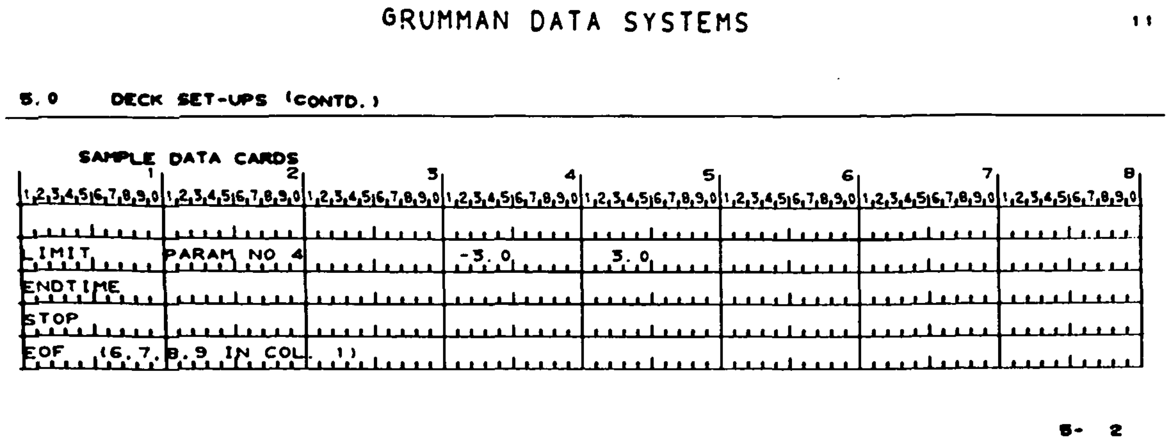
5. 1 DATA DECK SAMPLE

THE SAMPLE SHOWN BELOW INDICATES A TYPICAL DECK SET-UP. NOTE THE FIRST
<u>ENDTIME</u> CARD AND THE NEW SET OF PRINT/LIMIT CARDS WHICH FOLLOW IT.
EXPLANATION OF SAMPLE DATA DECK
TIME SLICE 1 (4/15/11.6 TO 4/16/22.3) WILL OUTPUT AS FOLLOWS -
PAGE 1 PARAM NOS. 1, 2, AND 3
PAGE 2 PARAM NOS. 4 AND 5
LIMIT CHECKING TO BE DONE ON PARAMS 2 AND 5
THE ABOVE OUTPUT GROUPS WILL BE RETAINED FOR TIME SLICES 2 AND 3 WITH TIME
SLICE 3 DOING, ADDITIONALLY, LIMIT CHECKING ON PARAMS 1 AND 3
TIME SLICE 4 (6/13/12. TO 6/13/30.) WILL OUTPUT AS FOLLOWS -
PAGE 1 PARAM NOS. 3, 7, 9, AND 2
PAGE 2 PARAM NOS. 5 AND 6
LIMIT CHECKING ON PARAM NO 4

SAMPLE DATA CARDS

1	2	3	4	5	6	7	8
1234567890	1234567890	1234567890	1234567890	1234567890	1234567890	1234567890	1234567890
TITLE			PAGE HEADI	NG			
TIME	04 15 11.6	04 16	22.3				
PRINT	10.	PARAM NO 1			PARAM NO 2		
PARAM NO 3							
LIMIT	PARAM NO 2		-133.64	4096.873			
PRINT	20.	PARAM NO 4			PARAM NO 5		
LIMIT	PARAM NO 5		-3.6	44.8			
TIME	05 01 03.6	05 01	25.4				
TIME	05 02 01.0	05 05	0.0				
LIMIT	PARAM NO 1		432.6	1230.5			
LIMIT	PARAM NO 3		-123.5	-23.6			
ENDTIME							
TIME	06 13 12.	06 13	30.0				
PRINT	20.	PARAM NO 3			PARAM NO 7		
PARAM NO 9			PARAM NO 2				
PRINT	5.	PARAM NO 5			PARAM NO 6		

Figure 1 (continued).–Illustration of documentation program.



GRUMMAN DATA SYSTEMS 11

5.0 DECK SET-UPS (CONTD.)

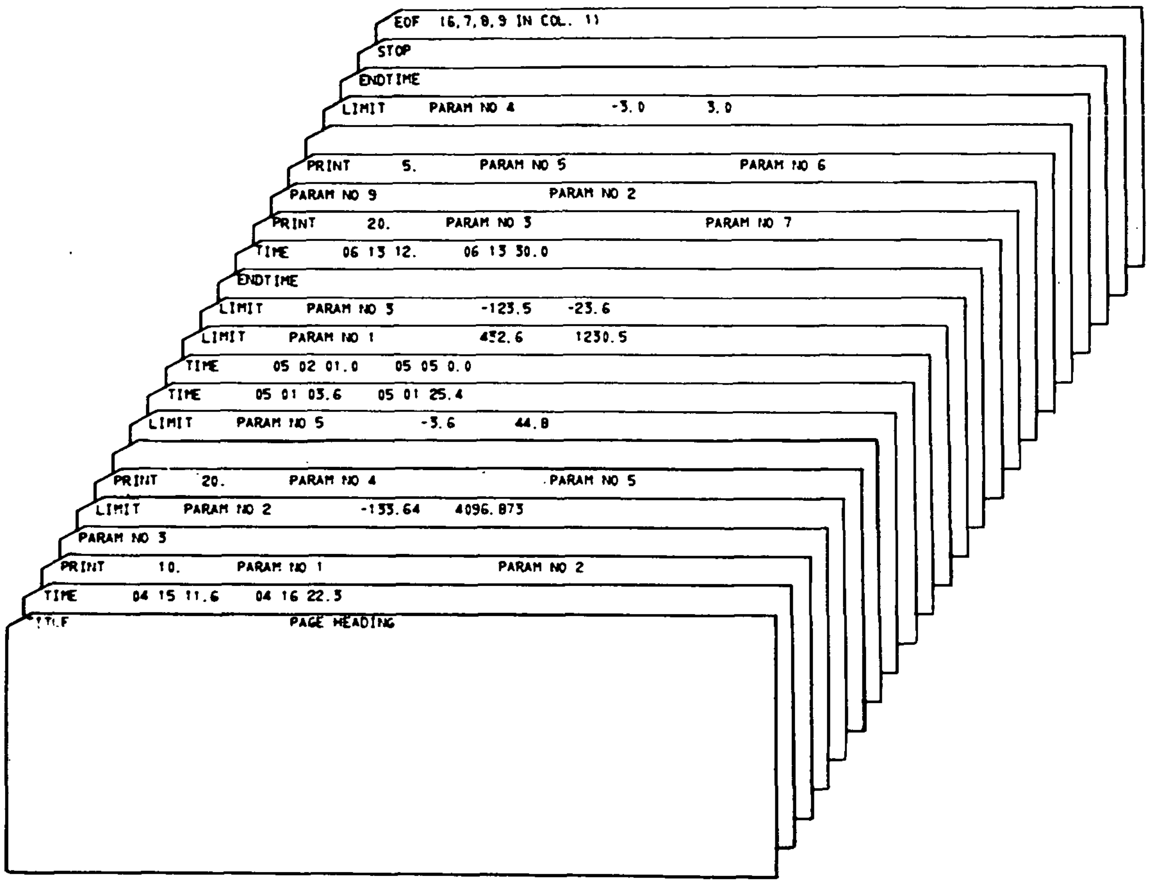
SAMPLE DATA CARDS

```
         1         2         3         4         5         6         7         8
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890

LIMIT     PARAM NO 4          -3. 0       3. 0
ENDTIME
STOP
EOF  (6.7.8.9 IN COL. 1)
```

5- 2

Figure 1 (continued).–Illustration of documentation program.

GRUMMAN DATA SYSTEMS 12

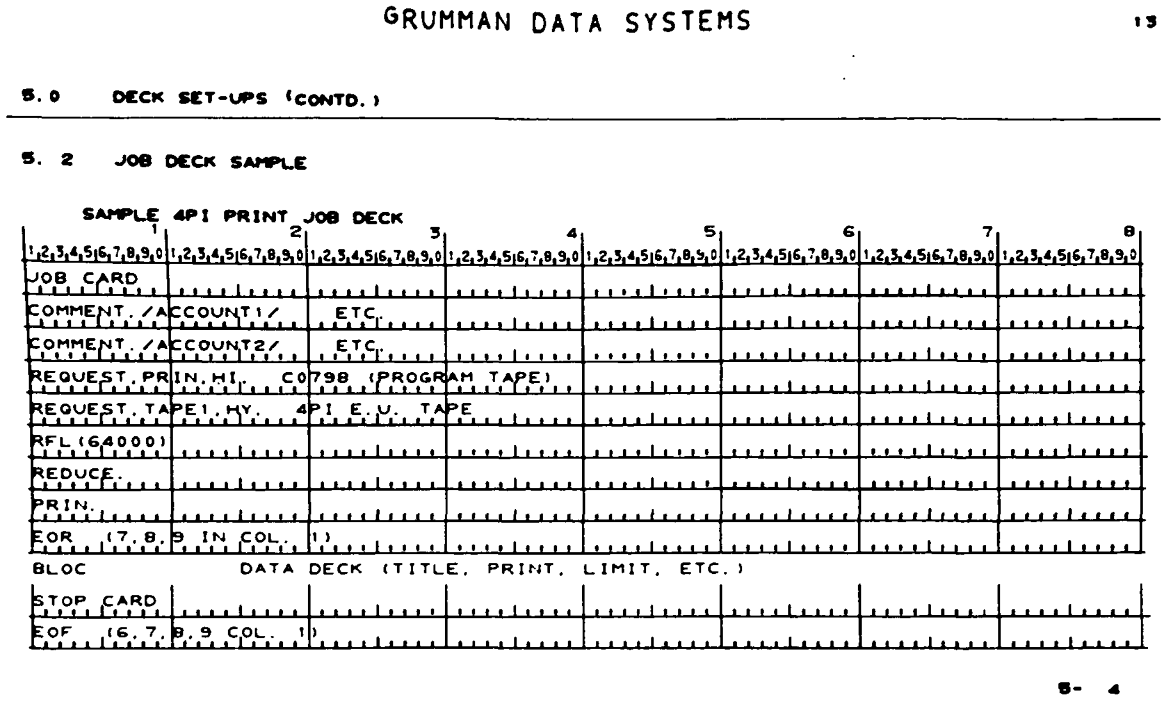


```
EOF     16,7,8,9 IN COL. 1)
STOP
ENDTIME
LIMIT     PARAM NO 4                -3.0        3.0

PRINT       5.        PARAM NO 5                         PARAM NO 6
PARAM NO 9                       PARAM NO 2
PRINT       20.       PARAM NO 3                         PARAM NO 7
TIME        06 13 12.      06 13 30.0
ENDTIME
LIMIT     PARAM NO 3                -123.5     -23.6
LIMIT     PARAM NO 1                452.6       1230.5
TIME        05 02 01.0     05 05 0.0
TIME        05 01 03.6     05 01 25.4
LIMIT     PARAM NO 5                -3.6        44.8

PRINT       20.       PARAM NO 4                         .PARAM NO 5
LIMIT     PARAM NO 2                -133.64    4096.873
PARAM NO 3
PRINT       10.       PARAM NO 1                         PARAM NO 2
TIME        04 15 11.6     04 16 22.3
TITLF                     PAGE HEADING
```

SAMPLE DATA CARDS

8- 3

Figure 1 (continued).–Illustration of documentation program.



5.0 DECK SET-UPS (CONTD.)

5.2 JOB DECK SAMPLE

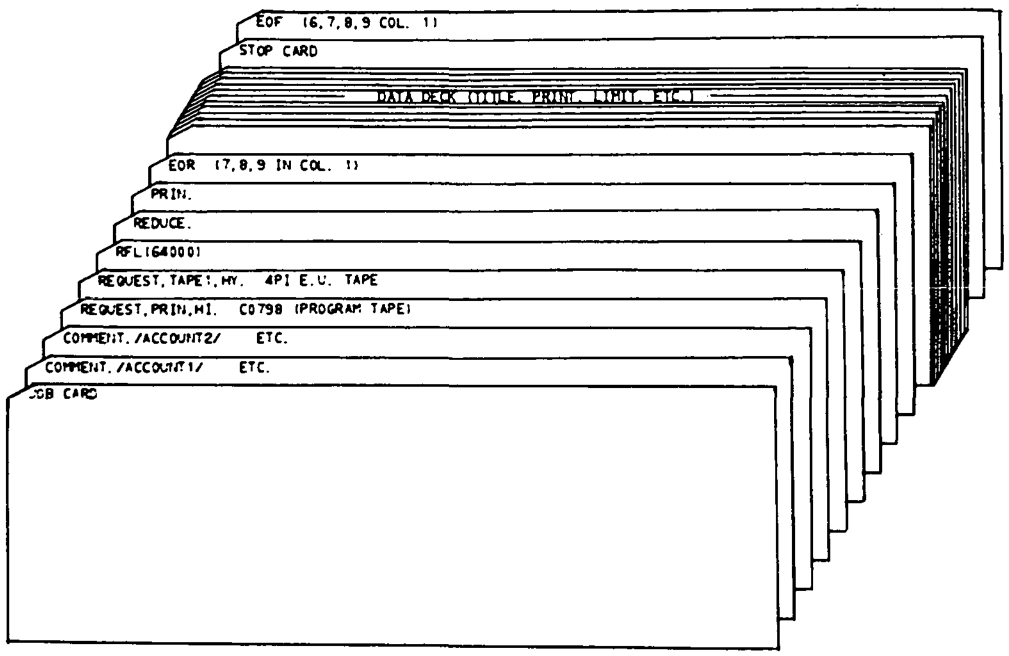
SAMPLE 4PI PRINT JOB DECK

```
JOB CARD
COMMENT./ACCOUNT1/     ETC.
COMMENT./ACCOUNT2/     ETC.
REQUEST.PRIN.HI.   C0798 (PROGRAM TAPE)
REQUEST.TAPE1.HY.   4PI E.U. TAPE
RFL(64000)
REDUCE.
PRIN.
EOR  (7.8.9 IN COL. 1)
BLOC              DATA DECK (TITLE, PRINT, LIMIT, ETC.)
STOP CARD
EOF  (6.7.8.9 COL. 1)
```

5- 4

Figure 1 (continued).–Illustration of documentation program.

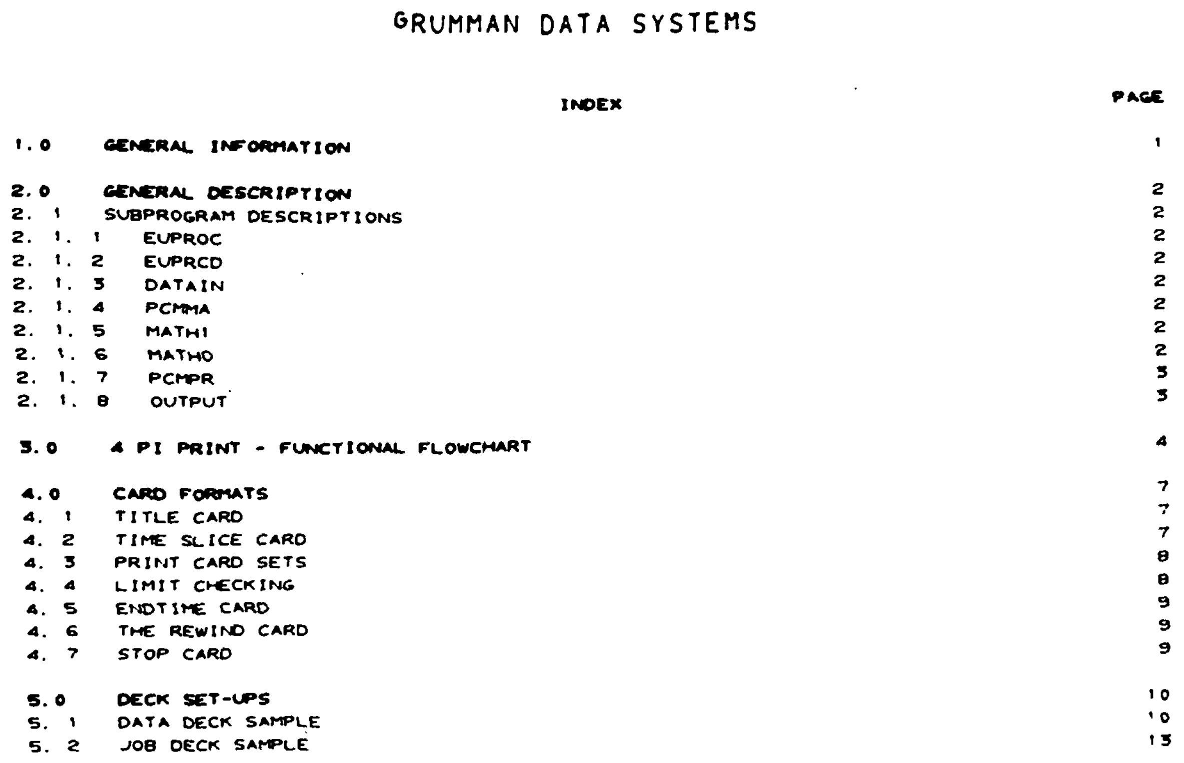
GRUMMAN DATA SYSTEMS 14



SAMPLE 4PI PRINT JOB DECK

5- 5

Figure 1 (continued).–Illustration of documentation program.



```
                              GRUMMAN DATA SYSTEMS

                                     INDEX                                PAGE

1.0       GENERAL INFORMATION                                               1

2.0       GENERAL DESCRIPTION                                               2
2. 1      SUBPROGRAM DESCRIPTIONS                                           2
2. 1. 1     EUPROC                                                          2
2. 1. 2     EUPRCD                                                          2
2. 1. 3     DATAIN                                                          2
2. 1. 4     PCMMA                                                           2
2. 1. 5     MATHI                                                           2
2. 1. 6     MATHO                                                           2
2. 1. 7     PCMPR                                                           3
2. 1. 8     OUTPUT                                                          3

3.0       4 PI PRINT - FUNCTIONAL FLOWCHART                                 4

4.0       CARD FORMATS                                                      7
4. 1      TITLE CARD                                                        7
4. 2      TIME SLICE CARD                                                   7
4. 3      PRINT CARD SETS                                                   8
4. 4      LIMIT CHECKING                                                    8
4. 5      ENDTIME CARD                                                      9
4. 6      THE REWIND CARD                                                   9
4. 7      STOP CARD                                                         9

5.0       DECK SET-UPS                                                     10
5. 1      DATA DECK SAMPLE                                                 10
5. 2      JOB DECK SAMPLE                                                  13
```

Figure 1 (concluded).—Illustration of documentation program.

DISCUSSION

MEMBER OF THE AUDIENCE: Would you comment further on the 26 keywords?

HANNEY: The topic, subtopic, and paragraph keywords are in the text generator. There is a card keyword for the coding form and a deck keyword for the deck display. The flowchart generator has 17 keywords, one for each symbol.

MEMBER OF THE AUDIENCE: Are the numerical displays used for text generation or simply stored on cards?

HANNEY: There are cathode ray tube (CRT) displays at the automated telemetry station, and we have a command program that interacts with the CRT unit. The user at this station can create a display code file and send it to the documentation programmer. It would then be stored on a disk file and inserted in the program later.

N73-19209

THE INTEGRATION OF SYSTEM SPECIFICATIONS AND PROGRAM CODING

William R. Luebke
Computer Sciences Corp.

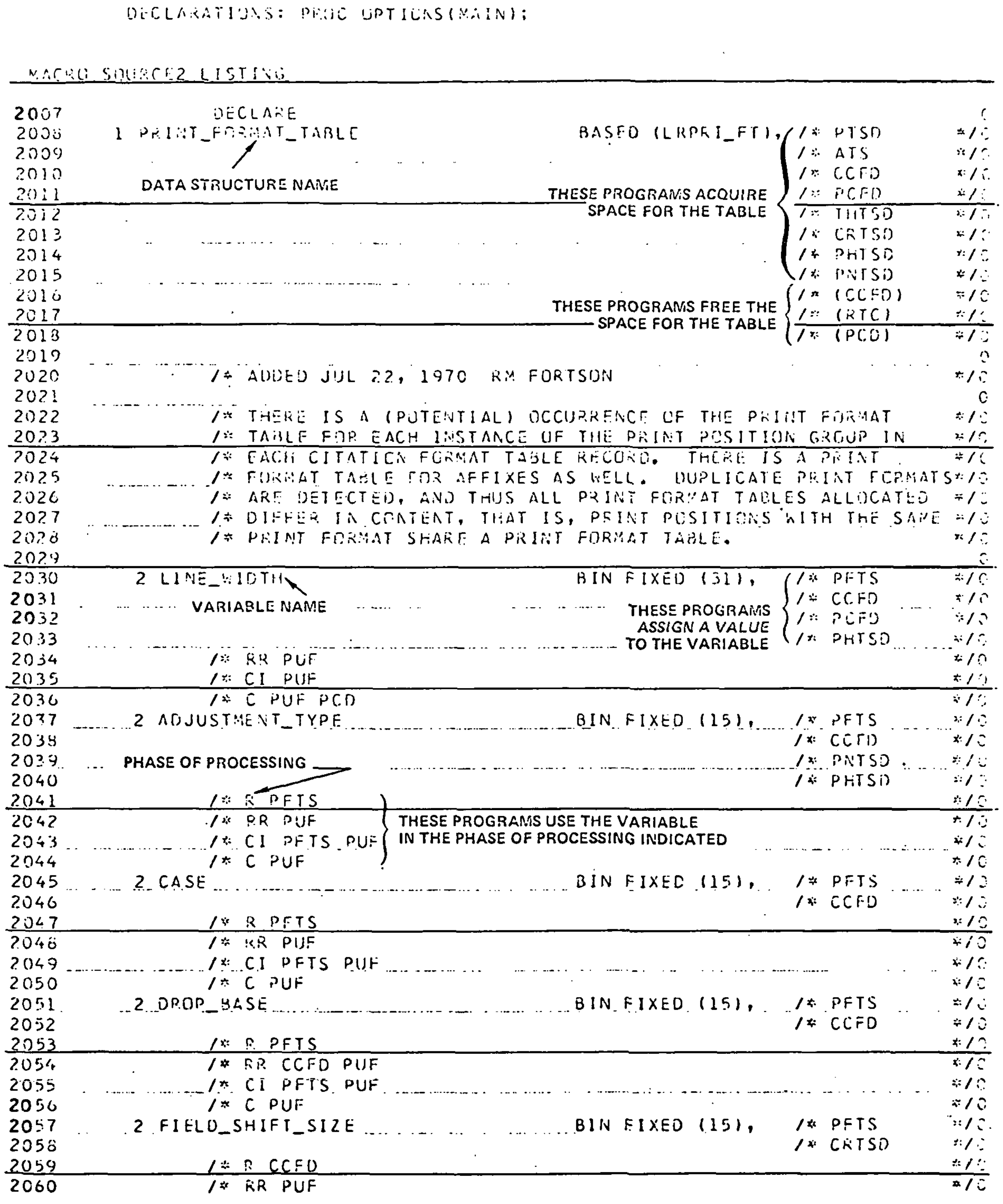
This report is a description of experience in maintaining up-to-date documentation for one module of the large-scale Medical Literature Analysis and Retrieval System II (MEDLARS II). Several innovative techniques have been explored in the development of this system's data management environment, particularly those that use PL/I as an automatic documenter. The PL/I data description section can provide automatic documentation by means of a master description of data elements that has long and highly meaningful mnemonic names and a formalized technique for the production of descriptive commentary. The techniques to be discussed are not common or unusual but are, instead, practical methods that employ the computer during system development in a manner that assists system implementation, provides interim documentation for customer review, and satisfies some of the deliverable documentation requirements.

DOCUMENTATION THROUGH DATA DEFINITION

MEDLARS II is a very complex system involving more than 50 PL/I programs that must share approximately 500 separate data variables. Most of the programmers assigned to the implementation phase had only limited experience in PL/I programming, and only a few had participated in the design of the system. The delivery schedule required that coding begin before the entire design had been completed and thoroughly reviewed. Therefore, an efficient mechanism for communicating the original design and any modifications of it to the programmers was essential.

With the number of programmers involved, each having responsibility for and knowledge of only a segment of the system, it was necessary to have ways to guarantee consistency in data definition and usage. The PL/I language has features that support these objectives. The data declaration statement in PL/I was chosen as the basis for design documentation to achieve data consistency, minimize coding and keypunching, and establish a basis for computer-produced portions of the final system documentation.

The DECLARE statement is a compiler instruction that defines the attributes and relationships of data variables. The data in the system are arranged in over 30 separate data structures or arrays of structures, each defined by a DECLARE statement. Figure 1 shows a portion of one of these structures, PRINT_FORMAT_TABLE. Line 1 names the structure and gives its attributes. The lines beginning with the number 2 name the data variables in the structure and establish their individual attributes. Comments in the PL/I language are


DATA STRUCTURE NAME
THESE PROGRAMS ACQUIRE SPACE FOR THE TABLE
THESE PROGRAMS FREE THE SPACE FOR THE TABLE
VARIABLE NAME
THESE PROGRAMS ASSIGN A VALUE TO THE VARIABLE
PHASE OF PROCESSING
THESE PROGRAMS USE THE VARIABLE IN THE PHASE OF PROCESSING INDICATED

```
DECLARATIONS: PROC OPTIONS(MAIN);

MACRO SOURCE2 LISTING

                 DECLARE                                                      C
   1 PRINT_FORMAT_TABLE                       BASED (LRPRI_FT), /* PTSD      */C
                                                                /* ATS       */C
                                                                /* CCFD      */C
                                                                /* PCFD      */C
                                                                /* THTSD     */C
                                                                /* CRTSD     */C
                                                                /* PHTSD     */C
                                                                /* PNTSD     */C
                                                                /* (CCFD)    */C
                                                                /* (RTC)     */C
                                                                /* (PCD)     */C
                                                                              C
           /* ADDED JUL 22, 1970   RM FORTSON                                */C
                                                                              C
           /* THERE IS A (POTENTIAL) OCCURRENCE OF THE PRINT FORMAT          */C
           /* TABLE FOR EACH INSTANCE OF THE PRINT POSITION GROUP IN         */C
           /* EACH CITATION FORMAT TABLE RECORD.  THERE IS A PRINT           */C
           /* FORMAT TABLE FOR AFFIXES AS WELL.  DUPLICATE PRINT FORMATS*/C
           /* ARE DETECTED, AND THUS ALL PRINT FORMAT TABLES ALLOCATED       */C
           /* DIFFER IN CONTENT, THAT IS, PRINT POSITIONS WITH THE SAME      */C
           /* PRINT FORMAT SHARE A PRINT FORMAT TABLE.                       */C
                                                                              C
     2 LINE_WIDTH                             BIN FIXED (31),   /* PFTS      */C
                                                                /* CCFD      */C
                                                                /* PCFD      */C
                                                                /* PHTSD     */C
           /* RR PUF                                                         */C
           /* CI PUF                                                         */C
           /* C PUF PCD                                                      */C
     2 ADJUSTMENT_TYPE                        BIN FIXED (15),   /* PFTS      */C
                                                                /* CCFD      */C
                                                                /* PNTSD     */C
                                                                /* PHTSD     */C
           /* R PFTS                                                         */C
           /* RR PUF                                                         */C
           /* CI PFTS PUF                                                    */C
           /* C PUF                                                          */C
     2 CASE                                   BIN FIXED (15),   /* PFTS      */C
                                                                /* CCFD      */C
           /* R PFTS                                                         */C
           /* RR PUF                                                         */C
           /* CI PFTS PUF                                                    */C
           /* C PUF                                                          */C
     2 DROP_BASE                              BIN FIXED (15),   /* PFTS      */C
                                                                /* CCFD      */C
           /* R PFTS                                                         */C
           /* RR CCFD PUF                                                    */C
           /* CI PFTS PUF                                                    */C
           /* C PUF                                                          */C
     2 FIELD_SHIFT_SIZE                       BIN FIXED (15),   /* PFTS      */C
                                                                /* CRTSD     */C
           /* R CCFD                                                         */C
           /* RR PUF                                                         */C
```

Figure 1.—Sample data declaration.

delimited by /* */. Notice that most of the DECLARE statement is comprised of comments. It is through the comments that the system's programs are related to the data.

The comments following the first line on the right side of the figure identify the programs that acquire storage or free storage (the latter indicated by parentheses) for the data structure. Below this is a comment paragraph that discusses the occurrence of the table within the system. Each data element in the structure is named on a line beginning with the number 2. Notice that long descriptive names are used in the declaration. PL/I permits up to 31 characters. At the right edge of a data-element line are comments that identify which programs set a value for that data element. For example, PFTS assigns a value to LINE_WIDTH. Below the data-element name are comment lines that identify the programs which use the data element's value. In the first case, the RR signifies one of the four phases of processing. In the RR phase, one routine (PUF) uses the element. In the C phase of processing, two routines (PUF and PCD) use the data element. The same type of information appears for all the data elements in all data structures.

Each programmer has a book of all the data structures used by the system. The book is produced by the PL/I compiler, and, in addition to the DECLARE statements, it contains an alphabetical listing of all data elements in the system. This listing, a normal compiler product, identifies the attributes of the element and the structure in which it appears.

The comments in the declaration are rigidly defined by position to permit simple manipulation of these data by a computer program. At present, a PL/I program processes all the declarations as input data and produces a listing of data involvement for each program. Figure 2 is an example of this program's output. The designer is now able to express any additions or modifications to the system design through changes in the declarations. These changes are quickly communicated to the programmer by means of the program data involvement sheets. Both the DECLARE statements and the program data involvement sheets will be part of the final documentation of the system.

There is another feature of the PL/I compiler that has been a very valuable aid in the use of the declarations. Before program compilation, a preprocessor scan is made of the compiler input, the source deck. The preprocessor phase permits inclusion of data from libraries in the system and certain procedural operations to take place before compilation. Presently, the declarations are being cataloged into the source library under a member name algorithmically derived from the name of the data structure. The preprocessor system is assigned two codes (LR and LP) to identify it within the system. LR is interpreted as data; LP, as procedure. The declaration statements are cataloged in the library with an LR prefix, followed by the first letter of each word in the name of the table. LRPFT is the library member name of the PRINT_FORMAT_TABLE. When programs are cataloged, their names are preceded by LP. For example, on the first line of the example declaration (fig. 2), it is indicated that the core area for the table was acquired by a routine named PTSD. In the library and in coding, the actual name of the routine is LPPTSD.

To gain access to the material in the source library, the programmer writes a %INCLUDE statement in his code, naming the member name of the table (e.g., %INCLUDE LRPFT). The member name is derived from the name of the table in the documentation. This statement will automatically cause the acquisition of both that library member and the declaration statement itself (from the source library) and will cause the declaration statement to be

```
PNTSD ALLOCATES SPACE FOR THE FOLLOWING TABLES

        1 PRINT_FORMAT_TABLE                  BASED (LRPRI_FT), /* PTSD


PNTSD ASSIGNS VALUES TO THE FOLLOWING PARAMETERS

        1 INITIALIZATION_STATUS_TABLE         BASED (LRINI_ST), /* CSP
          2 PRESENT_TABLE_ADDRESS             PTR,              /* PNTSD
          2 COSMIS_GROUP_SUBSCRIPT            BIN FIXED (31),   /* PTSD
          2 ABSOLUTE_PAGE_NUMBER_GROUP        BIN FIXED (15),   /* PNTSD
          2 COSMIS_SUBTABLE_TYPE              BIN FIXED (15),   /* PTSD
        1 MASTER_TABLE                        BASED (LRMAS_T),  /* CSP
          2 PRESENT_PAGE_NO_AREA_SIZE         BIN FIXED (15),   /* PNTSD
          2 PRESENT_PAGE_NO_SIZE              BIN FIXED (15),   /* PNTSD
          2 PRESENT_PAGE_NUMBER               BIN FIXED (15),   /* PNTSD
          2 PRESENT_PAGE_VERSO_INDICATOR      CHAR (1),         /* PNTSD
        1 PAGE_COMPOSITION_TABLE              BASED (LRPAG_CT), /* PFD
          2 PAGE_NUMBER_PREFIX_ADDRESS        PTR,              /* PNTSD
          2 RECTO_PAGE_NUMB_PR_FOR_TAB_ADD    PTR,              /* PNTSD
          2 RECTO_PREF_PR_FOR_TAB_ADD         PTR,              /* PNTSD
          2 VERSO_PAGE_NUMB_PR_FOR_TAB_ADD    PTR,              /* PNTSD
          2 VERSO_PREF_PR_FOR_TAB_ADD         PTR,              /* PNTSD
          2 PAGE_NUMBER_PREFIX_SIZE           BIN FIXED (15),   /* PNTSD
          2 CITATION_SEGMENT_NUMBERING_IND    CHAR (1),         /* PNTSD
        1 PRINT_FORMAT_TABLE                  BASED (LRPRI_FT), /* PTSD
          2 ADJUSTMENT_TYPE                   BIN FIXED (15),   /* PFTS
          2 FIRST_LINE_DROP                   BIN FIXED (15),   /* PFTS
          2 FIRST_LINE_SHIFT_SIZE             BIN FIXED (15),   /* PFTS
          2 FONT_ENTRY_NUMBER                 BIN FIXED (15),   /* PFTS ATS
          2 HEIGHT                            BIN FIXED (15),   /* PFTS ATS
          2 SHIFT_BASE                        BIN FIXED (15),   /* PFTS


PNTSD USES THE FOLLOWING TABLES AND VALUES

        1 INITIALIZATION_STATUS_TABLE         BASED (LRINI_ST), /* CSP
          2 COSMIS_SUBTABLE_REC_ID            BIN FIXED (31),   /* RID
          2 COSMIS_SUBTABLE_REC_ID            BIN FIXED (31),   /* RID
        1 MASTER_TABLE                        BASED (LRMAS_T),  /* CSP
          2 COMMON_POINTER                    PTR,              /* CSP
          2 PAGE_COMPOSITION_TABLE_ADD        PTR,              /* (PFD)
        1 PAGE_COMPOSITION_TABLE              BASED (LRPAG_CT), /* PFD
          2 RECTO_PAGE_NUMB_PR_FOR_TAB_ADD    PTR,              /* PNTSD
          2 RECTO_PREF_PR_FOR_TAB_ADD         PTR,              /* PNTSD
          2 VERSO_PAGE_NUMB_PR_FOR_TAB_ADD    PTR,              /* PNTSD
          2 VERSO_PREF_PR_FOR_TAB_ADD         PTR,              /* PNTSD
          2 OTHER_PAGE_COLUMN_LENGTH          BIN FIXED (31),   /* CFI
          2 FRPC_FIRST_LINE_WIDTH             BIN FIXED (15),   /* PCFD
          2 FIRST_LINE_OF_FRPC_PREC_BLK_WID   BIN FIXED (15),   /* PCFD
          2 VERSO_RECTO_INDICATOR             CHAR (1),         /* SFID
        1 PRINT_FORMAT_TABLE                  BASED (LRPRI_FT), /* PTSD
          2 SHIFT_BASE                        BIN FIXED (15),   /* PFTS
        1 FORMATTED_PRINT_UNIT_TABLE          BASED (LRFOR_PUT) /* PUF
```

Figure 2.–Program data involvement.

```
INCLUDED TEXT FOLLOWS FROM DD.MEMBER =  SYSLIB  .LRPFT

  % DCL (LIN_W, ADJ_T, CAS, DRO_B, FIE_SS,
           FIR_LD, FIR_LSS, FON_EN, HEI, LIN_D,
           MAX_NOL, OTH_LSS, SHI_B, PRI_FT, UNB_VI,NEW_SI) CHAR;
        % PRI_FT = 'PRINT_FORMAT_TABLE';
        % LIN_W = 'LINE_WIDTH';
        % ADJ_T = 'ADJUSTMENT_TYPE';
        % CAS = 'CASE';
        % DRO_B = 'DROP_BASE';
        % FIE_SS = 'FIELD_SHIFT_SIZE';
        % FIR_LD = 'FIRST_LINE_DROP';
        % FIP_LSS = 'FIRST_LINE_SHIFT_SIZE';
        % FON_EN = 'FONT_ENTRY_NUMBER';
        % HEI = 'HEIGHT';
        % LIN_D = 'LINE_DROP';
        % MAX_NOL = 'MAXIMUM_NUMBER_OF_LINES';
        % OTH_LSS = 'OTHER_LINE_SHIFT_SIZE';
        % SHI_B = 'SHIFT_BASE';
  % NEW_SI = 'NEW_SEGMENT_INDICATOR';
        % UNB_VI = 'UNBREAKABLE_VALUE_INDICATOR';
```

Figure 3.—Preprocessor statements.

physically inserted into the source code prior to compilation. All data declarations exist in only one place, the source library. All programmers use the same declaration from the library, and, therefore, changes in the data declaration will be automatically and immediately available to all programmers because the latest version appears in their program listing. In this way, data names and table identification are consistent.

Using the full 31 characters for naming a variable is awkward for a programmer and can lead to keypunching problems. The PL/I compiler helps here as well. In the preprocessor pass, before the actual compilation, it is possible to replace items in the source code. This feature is used to convert the programmer's abbreviation of a data variable name to the full name. A standard algorithm is used for abbreviating a name in a table: The first three characters of the first word, an underline character, and then the first character of each succeeding word. For example, the programmer writes line width as LIN_W, adjustment type as ADJ_T, and case as CAS. When he receives the computer listing, all the abbreviated names will have been replaced, and his listing will include the full descriptive name that was used in the declaration of the tables. The preprocessor code required to accomplish the conversion from abbreviated name to full name has been included as an addition to the member in the source library that contains the declaration of the table (fig. 3). Therefore, when the programmer includes the table declaration in his program, he is also including the preprocessor code that will accomplish all the abbreviation conversions for the data elements in the table. As a result, the preprocessor code is only prepared once, and the conversion is automatically performed any time the table is used. This does not mean, however, that the programmer cannot use the full version of the name. Either version can be used in this code.

IMPLEMENTATION

One man-week was required to code and catalog the declarations for the system library. This figure does not include the design of the structures or the keypunch time. Currently, table maintenance requires about 1/2 man-week for each calendar week. When system implementation began, 15 man-weeks were expended in coding and testing utility routines that output the data structures during program debugging.[1] This coding exercise was useful for training non-PL/I programmers and as an introduction to the rather complex data hierarchy.

PROBLEMS ENCOUNTERED

The basic concepts set forth in this paper have been validated by experience, and programmers find the structures easy to use. Design inconsistencies and program specification shortcomings appear to surface early in implementation, as soon as the program data involvement is available. However, if the reader intends to use this approach he should be aware of the following potential problems:

(1) The name abbreviations in the preprocessor statements must be unique. If they are not, the program using the declaration will not pass the preprocessor phase of compilation. This problem has caused several days of table lockout in our implementation.

(2) The printout material used by programmers must be updated with every non-comment change to the declarations. Failure to do this will cause the programmer to think that he has an error when his program is actually correct.

(3) A change in a data structure requires that all programs using that structure be recompiled.

(4) Program testing must be suspended while the data structures and related print programs are being updated in the library.

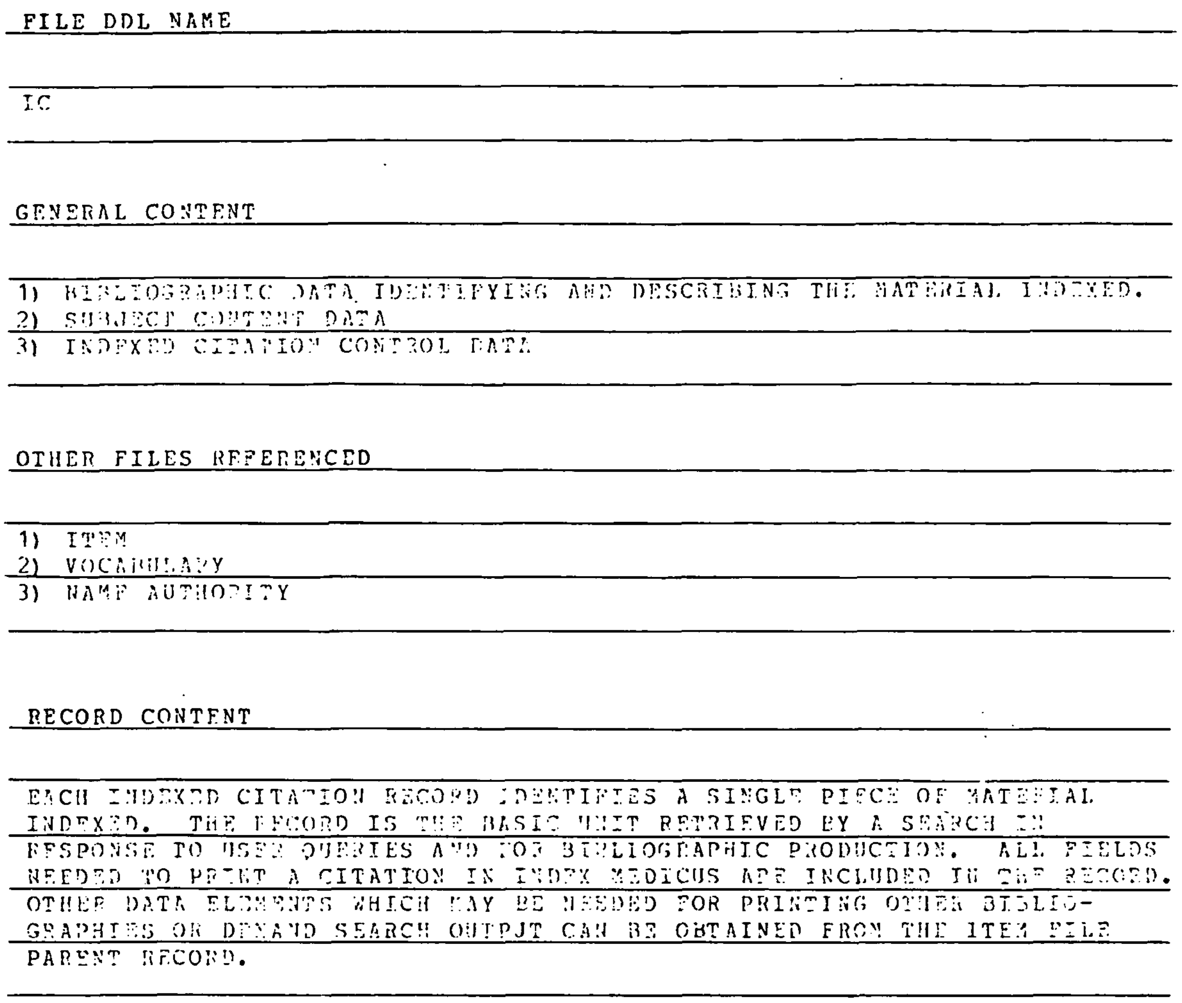
We found it best to update the tables no more frequently than once a month, to use extreme care, and to anticipate the sacrifice of at least one computer run by every programmer.

DATA FILE DOCUMENTATION

The remainder of this report briefly describes the manner in which user-oriented data files are documented. Approximately 30 individual files make up the MEDLARS II data base. A consistent method is needed for the definition of information carried in these files, a method that could be understood by non-data-processing personnel in the user's facility who are interested in the content, but not the structure, of the file and need a great deal of information about the files.

Nine descriptors were devised for every data element in a file, and these files were documented in machine-readable form so that they would be easy to update while the client thought about problems and requested changes. The files have been evolving, and the documentation has been able to keep abreast. Generally, it takes only a few days to document very sweeping changes in the design of the National Library of Medicine's bibliographic files.

[1]The structures are based on, and therefore cannot be output by, the PUT DATA instruction nor can the variables be traced with the CHECK function.



```
FILE DDL NAME

IC

GENERAL CONTENT

1) BIBLIOGRAPHIC DATA IDENTIFYING AND DESCRIBING THE MATERIAL INDEXED.
2) SUBJECT CONTENT DATA
3) INDEXED CITATION CONTROL DATA

OTHER FILES REFERENCED

1) ITEM
2) VOCABULARY
3) NAME AUTHORITY

RECORD CONTENT

EACH INDEXED CITATION RECORD IDENTIFIES A SINGLE PIECE OF MATERIAL
INDEXED.  THE RECORD IS THE BASIC UNIT RETRIEVED BY A SEARCH IN
RESPONSE TO USER QUERIES AND FOR BIBLIOGRAPHIC PRODUCTION.  ALL FIELDS
NEEDED TO PRINT A CITATION IN INDEX MEDICUS ARE INCLUDED IN THE RECORD.
OTHER DATA ELEMENTS WHICH MAY BE NEEDED FOR PRINTING OTHER BIBLIO-
GRAPHIES OR DEMAND SEARCH OUTPUT CAN BE OBTAINED FROM THE ITEM FILE
PARENT RECORD.
```

Figure 4.–Overview of indexed citation file.

Figure 4 shows the overview of the indexed citation file. The overview describes the general content of the file, identifies other files in the system that are referenced by this file in some way, and then generally describes the record content. The record content is just an overview that is used for a quick introduction to the file. It is followed by a description of the data within the file. Not shown is a pictorial representation of the file structure identifying all the data elements in the file. Each data element in a record is documented by punched cards that are numbered to identify the type of information carried. Figure 5 describes the numbered data-element listing. There is also a type of card for comments. Several small PL/I programs for formatting and editing the data have been written. A great deal of supporting keypunch work is required for implementation of the file, but once the file has been established, the data are easy to update.

```
1. NAME
     PUBLICATION MONTH
     FIELD NAME IN ABBREVIATED FORM:   PUBMO

2. DEFINITION
     THE YEAR AND MONTH DURING WHICH A RECORD IS FIRST AVAILABLE FOR
     FORMAL PUBLICATION.  IF A CITATION MUST BE REVISED AND REPUBLISHED,
     THIS FIELD WILL BE UPDATED TO SHOW THE LATEST DATE THE CITATION WAS
     AVAILABLE FOR FORMAL PUBLICATION.

3. PURPOSE
     USED FOR COLLECTING CITATIONS BY PUBLICATION MONTH FOR INDEX
     MEDICUS AND OTHER MONTHLY PUBLICATIONS.  USED AS A CRITERION FOR FILE
     SEGMENTATION AND RETRIEVAL.  A DIRECTORY IS PROVIDED.

4. OTHER RELATED FIELDS
     STATUS GROUP DATA ELEMENTS

5. STRUCTURE
     11 BIT BINARY FIELD -- YYMM.  THE YEAR IS CONVERTED TO BINARY AND
     STORED IN THE SEVEN HIGH ORDER BITS.  THE MONTH IS CONVERTED TO BINARY
     AND STORED IN THE FOUR LOW ORDER BITS.

6. VALIDATION
     NONE

7. INPUT SOURCE
     SET BY THE SYSTEM FROM THE STATUS GROUP.  THE USER MUST NOTIFY THE
     SYSTEM WHEN A CITATION IS COMPLETE AND READY TO PUBLISH.  THE SYSTEM
     WILL THEN GENERATE A VALUE FOR THIS FIELD WHICH SHOWS THE MONTH
     IN WHICH THE CITATION IS TO APPEAR IN ONE OR MORE FORMAL PUBLICATIONS.

8. FILE CONVERSION DATA
     NONE.    THIS IS A NEW DATA ELEMENT FOR MEDLARS II.

9. SYSTEM START-UP INFORMATION
     REQUIRED FOR SYSTEM STARTUP.
```

Figure 5.—Data-element listing.

DISCUSSION

MEMBER OF THE AUDIENCE: Do you feel that it is a worthwhile effort to attempt to document large programs that are continually changing?

LUEBKE: I think that it is a good idea and should be helpful, as part of the source documentation, when the system is finally operational. Having the data within the system catalog will permit programmers who did not participate in the development of the system to identify the relationships between the programs and the data so that they can perform maintenance.

MEMBER OF THE AUDIENCE: How many programmers are involved in your project?

LUEBKE: At the present time, we have about 27 programmers working in this area.

MEMBER OF THE AUDIENCE: For how many years has your project been in existence?

LUEBKE: We began programming in August 1967 and should be finished with that phase in the spring of 1971.

PANEL DISCUSSION

MEMBER OF THE AUDIENCE: I feel one of the most pertinent things that Dr. Swift said was that using automatic techniques to describe data is one of the most feasible and valuable things that can be done. In this last paper, there seems to be a suggestion of how to do that. Does the panel agree or disagree, and what are the relative merits of this approach?

PANEL MEMBER: I agree fully that the independent description of data is something that can now be done.

PANEL MEMBER: Not only description of data but also typing the description of the data to the system itself so that the data description entering into the system are basically the same ones the programmer works from.

PANEL MEMBER: I would like to add that besides describing the data, there is a problem of analyzing it. It becomes important to know where the data are throughout the programs and subprograms and how the data interact with each other after being changed.

PANEL MEMBER: My feeling is that as serious a problem as the traditional after-the-fact documentation is, the most serious part of the problem comes at the beginning of a project, what I call the "upstream documentation." This is the attempt to record information that everybody can understand, work with, and write code from. A considerable amount of documentation is brought into existence by going through the various stages of system design, from the requirements at the start of the project to the various elements and levels of the design approach to, finally, the stage that can be solved by writing code rather than by developing a further level of design.

I think the data processing business has perhaps been somewhat deficient in not putting enough emphasis on the total process of supporting and facilitating development of this upstream documentation. If properly done, it should form a body of material that can be coupled to the kinds of tools that are capable of being developed now for automatic documentation. This upstream documentation, rather than the problem of building from something that is already computer processable, is the basic problem.

PANEL MEMBER: I agree. Until program documentation flows naturally out of the designing process, it is always going to be a problem. You have to start at the beginning and generate most of the documentation in the process of designing a system. Until proper procedures are followed, good documentation is never going to happen.

PANEL MEMBER: Let me add that managers must also face the fact that certain costs are going to be incurred and that their project reports are going to reflect cost increases before any code will be written.

PANEL MEMBER: That comment compares the cost of getting the deck as it comes out of the computer without any comments to the cost of getting it with BELLFLOW. I think the manager should compare what it costs to put BELLFLOW comments in against what it costs to put in the comments that should be put in the deck anyway. Often if you insist on doing the job correctly in the first place, some of these other additions do not add much of a burden.

MEMBER OF THE AUDIENCE: The comment that programmers are undisciplined has been made several times. I wonder what the panel thinks about this.

PANEL MEMBER: I have a few random thoughts on the subject. One of the roots of the problem lies in programmer education. When programmers are trained, documentation very often is not stressed. No organized way to write a program is taught. Programs consequently reflect the idiosyncrasies of the programmer. But the organization of a program, given the same kind of problem, should be standard. Unfortunately, it is not. The solution to this part of the problem lies in better training.

Second, I agree that the only way to get reasonably good documentation is to have the documentation developed with the programming. For one thing, it is the only way it will get done because programmers are generally working on another project soon after the completion of one and have neither the time nor the inclination to document once the program is finished. In addition, they may forget certain aspects of the program. So, the only way to document is to integrate documentation with normal programming activity.

MEMBER OF THE AUDIENCE: I think you have to recognize that a computer program is a very difficult thing to describe in the first place. No system tells how to read documentation, there is nothing like a flowchart of how to proceed through a particular piece of documentation, which may be in narrative form.

The programmer works directly with a program and has no way of viewing it as a reader. I think one of the basic problems in program documentation is that programmers are not trained to think of how to make their programs readable to others. Most scientists, let alone most programmers, are not trained to write, and this fact must be recognized as we attempt to develop tools for automatic documentation.

Finally, the symbols that we use in flowcharts do not fit together with meaningful ways of expressing this information. I think that something ought to be done if we are going to have large names in data and procedure names that are 31 characters long. I wonder what the panel's comments on this are.

PANEL MEMBER: The situation in data processing is that the tools of documentation and description for computer programs that have been used are reasonably appropriate for basic computer program and data processing situations. But they are not really suitable for establishing and describing things like multiple-application, multiple-user, on-line, and real-time systems of various kinds. It is in many ways rather surprising that we have been able to do as well as we have in describing some of these newer situations with tools that were developed for an earlier kind of problem. Herein lies one of the main challenges for documentation.

PANEL MEMBER: I think one important question would be determining the amount of documentation that has to be done by people and the amount that can be done by computer.

PANEL MEMBER: When a building, the hardware part of a computer system, or a communication switching network is documented, something that can be seen and either agreed or disagreed with is being documented. Documenting a program is documenting an idea rather than something physical. It is more difficult to accept the fact that we have to spend money to figure out how to document and convey an idea to somebody than it would be for something physical.

You could show an executive director of a company, who has not been involved in a program, documentation of that program, and he would find it very difficult, even if it were the best documentation possible, to decide whether it is worth doing or not.

PANEL MEMBER: I think documentation should be divided into three stages. The first stage would be documentation of the planning stage. Then there would be documentation of the implementation effort. Once the program is implemented, there would be terminal documentation. Possibly, this might be the way to approach documentation. I would say that the motivation to document is certainly very strong at the beginning of a project. By looking at documentation in terms of a kind of life cycle, we certainly would get a great deal of this documentation completed at the very beginning. It may be that a lot of our thinking is simply not recorded at the time when it would be easiest.

PANEL MEMBER: I would like to comment on documentation of the implementation design effort. We are trying to record the development of a program so that if modifications are to be made to a program, there will be some idea of how complex the procedure of changing the program will be.

MEMBER OF THE AUDIENCE: One of the problems in documentation is that poor programs are often written. Very little is known about how to write programs. If you keep a program long enough, debug it long enough, and patch it long enough, it finally does everything it is supposed to do. Then, we often try to document these programs, which should not have been written in the first place.

PANEL MEMBER: I agree with you in one sense, and in another sense I do not. Very often what happens when programs are developed is that they are developed for certain specifications. The program, however, may be used for many years. If a little more effort were put into the program and a certain amount of flexibility were built into the program, then as the program changed, there would be less of a problem in adding to the program. I know it is difficult to try to foresee what changes may be needed, but I think if, for instance, the initial investment were increased by 25 percent, more than 100 percent may be saved during the life of that system.

PANEL MEMBER: Another problem is that finiteness is gradually disappearing, especially from the larger data-driven systems. There are now so many alternatives that to decide what future alternatives are is almost impossible.

PANEL MEMBER: Documenting programs that should not have been written is a problem. The programmer is not always at fault, however. Often he receives a specification that does not reflect the final product.

One company has eliminated this particular hazard by using intermediate personnel between the engineers and the programmers. They are familiar with the engineer and his field and also understand something about programming. These intermediaries read the specifications and translate them for the programmer.

As far as specifications are concerned, the person who wants the program written should know what he wants it to do. It is not enough to know what the input is and what the output should be. Good preliminary documentation is needed to write up a correct specification for a programmer. You have to have good preliminary documentation before you can give a specification to the programmer because he does what the specification tells him to and that is all he can do.

MEMBER OF THE AUDIENCE: After a program is debugged, it is assumed to operate perfectly. That is not true about hardware. In hardware, fault indicators and other safeguards are built in so that if a system does not operate properly it stops. The equipment is often capable of indicating why the machine is failing. Documentation certainly is often used to try to track down problems, but not to the extent it could be. The typical commercial application is developed so that you can check for data errors, but the program will not check to see that it itself is performing properly.

PANEL MEMBER: Many of the things we do, of course, have been done by the systems approach but because of various factors including the undependability of Government funding, we often start and then stop programming efforts. Another problem is that badly documented programs are often inherited.

MEMBER OF THE AUDIENCE: This brings up an interesting point. I think all the panel members have operated under Government contract. What do you think of the specifications laid down for programming documentation and the followup action on the part of the Government from the start of a program to the delivery of the final product?

PANEL MEMBER: I believe that all contracts should stipulate that the contractor maintain and document his program for a specified length of time after the completion of the contract.

MEMBER OF THE AUDIENCE: How do you handle the problems of having to take over a system that is already set up and documenting for others?

PANEL MEMBER: One case that I happen to know about concerned a defense-oriented system of considerable size. It became necessary for another group to take over its maintenance. In that particular instance, the company worked backwards. They began by specifying what the requirements were when the requirements specification was developed. They then proceeded to go through the first level of developing the end-item specification, called the general design specification. These appeared to be in agreement with what was going on. Then the next levels of specifications were produced and checked until, finally, all the specifications that ought to have been produced in the process of developing the program in the first place were written. Some "fudging" brought the actual code into agreement with the specifications thus created. Only at that point did it really become possible for the company to relax and begin actual maintenance.

MEMBER OF THE AUDIENCE: One of our principal customers has a requirement that many of our programs be sent to COSMIC for further distribution. In the past, we had a great deal of difficulty in getting the programs accepted. We solved that problem by setting up a review team independent of the original programming group. The review group is composed of operations, engineering, and programming people that take a program and work through the program library, program by program, to see whether it is completely understandable and can be shipped out to someone else with the assurance that it works and can be run elsewhere on the same machine.

The original programming group has a fixed budget for each program sent through the review group. If it costs more than the allotted amount to review, the originating cost center gets tabbed for the excess cost. We have not had enough experience to see how this is going to work out, but this factor of economic accountability would seem to guarantee its success.

MEMBER OF THE AUDIENCE: The talk about developing the documentation for the life cycle of a program sounds very nice; however, for some programs that is not necessary. I think automation can really be useful by keying information on data in different ways.

I think there should be a file of information about a program that can be called upon when necessary, but there is no need for reams of information that you cannot find your way through. The trouble is not having enough documentation but having so much that you cannot begin to understand how to use it or how to be able to get into it. There are times when you have to change a program in a short period of time. Then you need a certain amount of assurance that you have documentation for and know the location of all the data of a certain kind for each program. You do not need to have something printed out every time one factor in a program is changed. Maybe we should look upon the computer as being an information retrieval system of documents and documentary information about various elements of the program and its logic and not let it become a producer of printed documents.

PANEL MEMBER: I agree that we do not need to print all the necessary documentation. We find a tape recorder very useful in documentation in two ways. General descriptions and information about the program are recorded but not printed. We keep the tape for reference only. Generally, the original designer talks about where you might change the program and certain idiosyncrasies of the program. This tape gives a very good picture of how the program was developed and why and where you might have to be very careful if changes have to be made.

We also use the cassette to record basic information that a programmer should write but never gets around to doing.

MEMBER OF THE AUDIENCE: If you had clear-cut specifications in advance, then clearly all you need is somebody to code it. That is one thing. But in advanced theoretical research, you probably cannot get clear-cut specifications in advance that will lead to efficient ways of eventually writing the program. I think that we may lose some creativity in programming if we force too many specifications on programmers.

PANEL MEMBER: I would say that 90 percent of all the programming done in the United States is not creative. I would also say that there is certainly no reason why one should not have complete flexibility in the other 10 percent of the cases.

PANEL MEMBER: If you are generating a system to be used only once, you do not care much about how it is evolved. If you are generating something that is going to be used many times, then you should not be so creative that you ignore efficiency and good design.

MEMBER OF THE AUDIENCE: You mentioned a 25-percent increase in cost to make programs easier to handle. How do you decide which programs will persist and thus need additional effort to make them meet future requirements?

PANEL MEMBER: In some situations, for example, a payroll system, it is relatively easy to see how the system might have to change in the future as the company changes. In other cases, for example, the space program, how the system will change is not so obvious but the fact that it will is. So some allowance for change has to be made and is worth the extra 25 percent. But even in these obvious cases, little is being done.

PANEL MEMBER: It seems to me that almost any program of reasonable size that is being developed is worth spending that extra 25 percent on. You may be wrong, and it may have only limited life. But if you think that it is going to have long life, you ought to put in extra effort.

Session II

15

N73-19210

VIEWS ON COMPUTER PROGRAM DOCUMENTATION AND AUTOMATION

Dr. Herbert R. Grosch
National Bureau of Standards

The topic of this talk is not really computer program documentation and automation. Because the Bureau of Standards is not very advanced in these fields, this paper will confine itself to comments about various aspects of the problem of documentation and description. The first aspect I will discuss is what has been called the problem of semantics.

Scientists have access to three levels of data. The first is the personal information that is often written on the backs of IBM cards or 3 by 5 cards: the kind of information that is used to make a particular project work. The second level, more formalized and organized, but usually not more mechanized, includes printed catalogs, formally kept laboratory note-books, and computer printouts with comments for a unique program. Finally, there are the mechanized data banks and the mechanically formalized documentation of the computer.

The topic of this symposium is how to transform the second level of information into the third level of information with a minimum of thought, time, and attention. This will be much more difficult than you may realize. Information has no independent life of its own. What it means to each of us depends on our individual backgrounds, training, and experience. The same set of figures that means a useful window for blastoff to an engineer on the ground means an extremely narrow passage for an astronaut who is going to have to squeeze through that same window.

Similarly, the information produced by documentation of a program, if presented to several different groups, may mean entirely different things to these groups. The solution to the problem of documentation, in my opinion, is not strict formatting or extensive use of standards. Nor is heavy investment in professional regularization of documentation and of the surrounding data the solution. Instead it is the recognition of the fact that people look at problems differently.

If this recognition were taken to an extreme, it could be argued that everybody should document his own work and do everything by hand. But that would be an impractical solution. After all, the purpose of the English language, of installation standards, and of programming manuals is to provide a common ground for the transmission and use of information. The goal of this symposium is to look for ways to provide this common ground quickly and inexpensively. This is an admirable goal. But unless work toward this goal is accompanied by a recognition of the differences between the ways men look at information, the goal will not be realized.

An example of the problem created by these differences can be seen in the field of software exchange. If a programmer tries to keep track of his system so that he can hand it on to a successor working in the same position for the same organization on the same computer and solving essentially the same problem with just a different set of input numbers, the chances are that there will be very few problems. One reason is that the information to be exchanged is done almost always between men of similar backgrounds and training, at least as far as this particular problem is concerned.

However, when a company decides to send a program to COSMIC so that it can be distributed to people with similar backgrounds and training to be used in solving a similar problem on similar equipment, the task of transmitting the same information is much more difficult, as most of you know. The paper presented by the COSMIC representative illustrates this perfectly.

Finally, if a person decides to go into the business of selling programs, i.e., transmitting programs to strangers working on different problems on different machines, the problem of documentation becomes almost insurmountable. If you talk to people who are in the business of selling and maintaining programs, you will find that perhaps 80 percent of their costs are essentially maintenance and documentation costs.

The prospects for using machine language alone in seeking a solution to the problems of documentation are quite discouraging. The problem with machine-independent language is that it fosters the illusion that it is possible to document without rethinking; i.e., without thinking about what needs to be changed to use a program in a new system or new machine.

The difficulty of documenting cannot be eliminated because documentation cannot be done automatically; thought must be involved. However, the automatic approach does have the important advantage that changes can be made easily while documenting. Automation streamlines the process of picking out and putting together the elements needed to document a particular program.

Another aspect of the problem of documentation is its cost. Nobody knows within 30 or 40 percent just what computing and data processing cost the Federal Government. One probable estimate is that computing and data processing, communications, and library science information technology with its information retrieval, documentation, and microfilm cost the Federal Government well over $10 billion a year. It costs the United States as a whole perhaps $50 billion a year.

One reason for this enormous cost is that all the computers, information retrieval devices, and other specialized devices operate at no more than 5 to 30 percent efficiency at best. This is certainly not very efficient.

Despite the fact that it is patently impossible to run anything at 100 percent efficiency, there are some ways of reducing costs by a factor of 2 or, keeping costs the same, increasing throughput by a factor of 4. This symposium is an attempt by the technical community to do this singlehandedly. It is doomed to failure because management and administration have not kept up with technological advances. To put it somewhat facetiously, the technologist makes things as good as they are, and the manager makes sure they do not get any better.

The kind of organizational and administrative change and the detailed management needed to see that technological advances are used is lacking not only in Government but also in business to a great extent. Moreover, it is almost impossible for technological improvements to be used properly. The principle by which people are promoted from technical positions to high-level managerial positions effectively excludes anyone with a real desire to achieve efficiency.

The only solution to this problem seems to be to attack the problem this way. There do exist large-scale, on-line systems that work because they have to. NASA is an example of one, the airline reservations systems are another, and the stock market is yet another. These systems are publicly visible and are subject to economic accountability. If the astronauts do not land alive, if an airline reservations system does not work properly, or if someone loses $50 million worth of stocks, then somebody will be fired. These systems should serve as models for the solution to the problem of how to keep our costs within reasonable bounds.

N73-19211

AUTOMATIC EDITING OF MANUALS

Dr. Robert P. Rich
Applied Physics Laboratory, Johns Hopkins University

Documentation for a computer program is usually understood to include some or all of the following items:

(1) Program listing
(2) Flowcharts
(3) Problem description
(4) Programmer's reference manual
(5) Analyst's reference manual
(6) User's manual
(7) Management information

The documentation problem that one encounters arises from the difficulty of getting all of these items prepared in a timely fashion and the near impossibility of keeping them all correct and mutually consistent during the life of the program.

A useful approach to the problem is to collect all of the necessary information into a single document, which is maintained with computer assistance during the life of the program and from which the required subdocuments can be extracted as desired.

Implementation of this approach requires a package of programs for computer editorial assistance and is facilitated by certain programming practices that are discussed in this paper. Experience shows that this approach not only provides documentation at a reasonable cost but also facilitates program implementation and management, especially for large programs requiring a team effort.

THE INFORMATION PACKAGE

The present approach to program documentation was made possible by the existence of a general-purpose information package for the management of files of textual material. This package was originally developed for document retrieval with the IBM 1401in 1962. It has been rewritten with successive improvements for the IBM 7094, CDC 3300, and IBM 360; this last version, INFO 360, is discussed in this paper. The package contains three major programs: the EDIT program for maintaining standard files, the PRINT program for printing a standard file, and the SEARCH program for selecting records from a standard file.

Preceding page blank

The Standard File

For a specific application, INFO 360 works on standard files. Each file consists of a number of standard records. Each record consists of a format code followed by a comma and then by the body of the record, which is a string of characters whose number cannot exceed 4000. The master file contains the text of the current draft of the document. Most of the records are text records, with a hyphen as format code:

-,This is a text record.

Another type of record that is important is the title record, with format code T*d*, where *d* is a single decimal digit specifying the level of the title:

T3, 37. Title at level 3.

These records are used as section headings within a document; the body of the record begins with a section number (37 in the previous example).

The EDIT Program

The EDIT program permits a master file to be established or modified in a way that is specified by a file of changes. One or more records may be inserted, deleted, or modified. The details of how the changes are specified are not of concern here, but it should be noted that a good secretary can learn to type the changes without much difficulty. The cross-reference feature of the EDIT program is important in the present application. As an example, consider the section numbers of title records. The EDIT program can be instructed to renumber all the sections in the sequence 1,2,3, . . ., throughout the file. It also corrects cross-references, replacing the old section number with the new one, so that cross-references by section numbers remain correct if the file is modified.

The PRINT Program

The PRINT program outputs a standard file in a variety of formats that are determined jointly by the format codes of the individual records and the values assigned to various print parameters. Text records are broken into lines (with hyphenation and justification) to fit the specified margins. The listing is broken into pages to fit the assigned page length; page headings and numbers are printed as requested; and footnotes and white space for figures are appropriately positioned, no matter how the page breaks fall. In addition, multicolumn printing may be specified.

The treatment of the title records is especially interesting. When the PRINT program encounters a title record, it uses this record in three ways:

(1) The body of the record is printed in the text at its point of occurrence and set off by lines of asterisks.
(2) The body of the record is saved as a page heading, to be repeated at the top of following pages until it has been replaced.
(3) The body of the record, with the current page number added, is saved for inclusion in the table of contents.

When printed, each title record is indented by an amount proportional to its level.

This treatment of title records by the PRINT program, together with the automatic correction of cross-references by the EDIT program, provides an extremely powerful cross-referencing mechanism for the complex type of document involved in the present application.

THE MONODOCUMENT

What is being proposed is that the complete documentation for a particular program be included in a single document, the monodocument for that program, and that this document be maintained with the assistance of a computer employing INFO 360 or any local equivalent. When such a program is begun, the monodocument consists of only an outline that may be in the form of title records at appropriate levels so that the indentions will preserve the correct outline form in the table of contents.

If several people are to work on the program, then the first coordination meeting might result in the assignment of responsibilities for the various sections. Each such assignment is recorded temporarily as the text of the section, to be replaced by the actual text when it becomes available. As each section is completed, it is put into the monodocument by the EDIT program; the section then becomes immediately available for proofreading and further correction by author or editor and for reference and negotiation by all members of the team. Once a section has been approved and proofread, it remains correct until changed. Hence, all people involved can concentrate on the currently active sections of the document.

As agreement is reached on such matters as file formats, subroutine specifications, and programming conventions, they are incorporated into the document. Since at any given time, each person is using a copy of the same draft, it is much easier to maintain consistency. Each programmer has the responsibility of ensuring that his part of the program remains consistent with the other relevant parts of the monodocument. The cross-reference capability makes this part of his job such simpler. If the monodocument grows in this way as the program is written, it will be completed when the program is completed, and the documentation problem will have been solved.

The Symbolic Program

The symbolic program itself is one of the major sections of the monodocument. It is easy to incorporate the symbolic cards for a checked procedure into the master file or to punch such cards from the master file. Hence, the monodocument can easily contain the program as it was checked out; in fact, it becomes the official record of the final version of the program. This is particularly helpful if the program has a long production life.

Although the general approach being discussed is fruitful for assembly language programs, the symbolic program is especially elegant when high-level languages are used because a moderate amount of commentary and proper style conventions make the program self-documenting in a very useful manner. It is assumed, of course, that the program is written in a modular fashion. Such helpful techniques as assignment of labels in lexicographic order, systematic indention to show logical levels, and explicit declaration of variables deserve more attention than is given to them in this paper. The fact that program commentary can contain cross-references to other sections of the monodocument is potentially very helpful but has not yet been exploited.

Flowcharts

A set of flowcharts is a traditional part of the program documentation package. Recent experience indicates that a program properly written in a high-level language, especially when cross-referenced to an appropriate functional description, is easier to understand than the flowcharts that purport to describe it. Those who hold this view would ignore flowcharts in the monodocument. (This opinion does not apply, of course, to the informal, working flowcharts that the programmer uses while he is writing the program.) However, if they are required, they can be prepared by AUTOFLOW after everything else has been done. Those who still feel that final, hand-drawn flowcharts are worth what they cost will, of course, continue to produce them; that job will not be made more difficult if the monodocument approach is used for the rest of the package.

Problem Description

Problem description means a description of the problem the program is to solve, the function it is to carry out. This section of the monodocument will obviously be one of the first ones actually written, although it may be modified from a brief qualitative description to a detailed technical specification in some instances as the work progresses.

Programmer's Reference Material

The programmer's reference material portion of the monodocument contains information of interest to a programmer who has to correct, modify, or explain the program. It is complementary to (and can be cross-referenced to) the symbolic program.

Analyst's Reference Material

The analyst's reference material includes details of interest to the user that are not included in the user's manual. For example, the convergence characteristics of the numerical algorithms and the nature of the approximations used are among the items included. Some material could equally well be placed here or in the programmer's reference material.

User's Manual

The user's manual includes the material needed to use the program: input formats, control cards, file designations, alarm messages, restrictions on ranges of input variables, etc.

Management Information

Management information, such as time logs of personnel assignments, computer time for checkout, and comparisons between original estimates and actual performance, can easily be kept current as the monodocument is periodically updated.

DISCUSSION

MEMBER OF THE AUDIENCE: What can be said of the system's ability to produce machine-readable charts, tables, and illustrations? Many document programs have a need for them.

RICH: Those displays that consist essentially of computer printout, such as AUTO-FLOW charts, could very easily be imbedded in the document and updated in the same manner as text. Those displays that are produced by a plotting device or by a draftsman would be incorporated into the document when it is bound.

MEMBER OF THE AUDIENCE: Some of the IBM systems that are available have languages for producing machine-printable charts, illustrations, and diagrams that are not flowcharts.

RICH: The system that I discussed, as well as a number of other systems, have features that were not covered in my presentation. For instance, our system will not only accept tables but will even perform the arithmetic of tabular work.

Currently, we have over 200 pages of documentation for our system. For instance, the PRINT program has approximately 30 different format codes that indicate the different types of records: text and heading records, indexing records, records that leave space for figures and captions, etc.

MEMBER OF THE AUDIENCE: How does your approach handle the engineering aspects, such as representation of algorithms in text? A very important part of scientific programming is the ability to display the equations being used.

RICH: There are essentially two solutions to this. Space can be left in the text so that typed equations can be stripped into the document. This is quite troublesome. I take the view, however, that if we are going to achieve the documentation of a program, the problem is best described in a programmable notation. If the engineer wants an alpha, then I spell ALFA; if exponents are required, then two asterisks should be used, or some other representation that would depend upon the programming language. If an algorithm is clearly defined for the computer and an appropriate language is being used, the engineer can easily verify his statements. It is best to help the engineer write his formulas in a programmable language.

MEMBER OF THE AUDIENCE: Is it your feeling that another individual, a documentation specialist, should be working with the programmer and handling all of the evolutionary documentation?

RICH: When I work on a program, I take personal responsibility for the documentation. For the usual systems team (project engineer, physicist, analyst, programmers, etc.), a secretary and an individual willing to take responsibility are needed in order to achieve good documentation.

MEMBER OF THE AUDIENCE: What are the advantages of using the computer for documentation instead of an MTST or similar device?

RICH: The computer generates text that is truly machine-readable; this is not always the case with MTST's. For updating text efficiently, the computer approach is far superior.

N73-19212

MAKING AUTOMATED COMPUTER PROGRAM DOCUMENTATION A FEATURE OF TOTAL SYSTEM DESIGN

Allan W. Wolf
System Development Corp.

The "paper-mill" character of large-scale computer software systems is a condition that is all too familiar to anyone involved in the computer programming business. Program manuals, design specifications, administrative reports, system descriptions, and user's manuals are just a few of the kinds of documents necessary to support a big software system. These documents, besides being complex, abundant, and subject to change, are frequently afterthoughts to the systems they support rather than part and parcel of the system design. This factor, when coupled with deadline and money pressures, unfortunately leads to another all too familiar condition–inadequate documentation.

Program documents are too often fraught with errors, out of date, poorly written, and sometimes nonexistent in whole or in part. This condition need not exist, however. Data stored on the printed page should be accurate, accessible, and helpful to the user, and it can be if a systems approach and existing computer technology are employed. This paper describes how many of these typical system documentation problems were overcome in a large and dynamic software project.

The project that will be discussed is the U.S. Air Force Satellite Control Facility (AFSCF) orbital prediction and command system. It is both large and dynamic and consists of about 2.5 million machine instructions in some 900 programs, over one-half of which are being modified during a typical 6-month period. The documentation supporting this system amounts to some 65000 pages, and an average of 5500 new and modified pages are published each month.

At one time there were numerous system problems that could be attributed to a lack of quality in the software documentation. More than 10 subcontractors produce the satellite computer program subsystems for AFSCF operations. It is common to have several of these agencies simultaneously produce programs that must interact (or interface) with each other and also with existing software. Before the development of the current system, this procedure led to problems in computer storage sharing, program calling sequence interpretation, interface data design, and other areas. In addition, there were all the usual problems associated with poor documentation.

(1) Users did not know how to call or use existing programs. Required input parameters and data could not be determined, and, in some cases, it could not even be determined whether routines existed to perform a particular function. Expensive computer time was wasted experimenting, or essentially duplicate routines were produced because it was felt to be cheaper than the process of decoding existing ones.

(2) Analysts could not determine whether existing routines were adequate for certain applications. References to the mathematical bases for programs were lost, or original work was never documented.

(3) Maintenance and development programmers spent much time and money attempting to modify programs. Great savings could have been realized if there had been adequate standards and conventions or system documents to provide some insight into what had already been done.

As the AFSCF system grew larger and more complex, and as these documentation problems manifested themselves in various ways, it became obvious that they would have to be solved, or the system would become totally chaotic. System Development Corp. (SDC) was given the task of designing and developing a new software system to support AFSCF. There were several design goals for the new system, but the one of primary interest for this paper was the alleviation of the types of problems emanating from inadequate documentation.

To fulfill this and the other design goals, a total systems approach was used. For documentation, this means that each system component was designed with the documentation problem in mind, instead of a procedure that designs the system and considers documentation as an appendage to be developed after and around the basic design. Naturally, there were compromises because of conflicts among the several goals, but the final result was a system that directly incorporated features in the basic design that overcame many of the previous documentation problems. The systems approach encompassed such items as—

(1) Configuration management (a closely monitored software management scheme that guides products through the various design, development, and acceptance milestones)

(2) Standards and conventions (guidelines, restrictions, and quality assurance measures covering many of the program design and development activities; application handled by configuration management)

(3) Collection of program information into central data banks to which all system components will have access, permitting easy and accurate documentation)

(4) Interaction among executive, compiler, central data banks, and configuration management (to provide a check and balance system that will prevent errors)

(5) Automatic documentation (to provide timely and accurate documents)

Figure 1, which will be discussed in detail in the section entitled "Configuration Management," shows the flow of new products through the milestones in the AFSCF system. This illustrates the interaction among some of the items just mentioned.

This paper shows how the system approach guarantees the accuracy of various portions of the documentation, provides the user with a total picture of the system, provides calling sequence and internal information on the various system programs, and, in general, eliminates many of the problems that typically arise from poor documentation. Specifically, the following elements will be discussed:

(1) The data-base definition, or common pool of information (COMPOOL), which is a data base in itself, contains descriptions for every program and piece of interface data (between programs) in the system. It plays a major role in the generation of automated documentation, the simplification of program maintenance, and the minimization of program development costs.

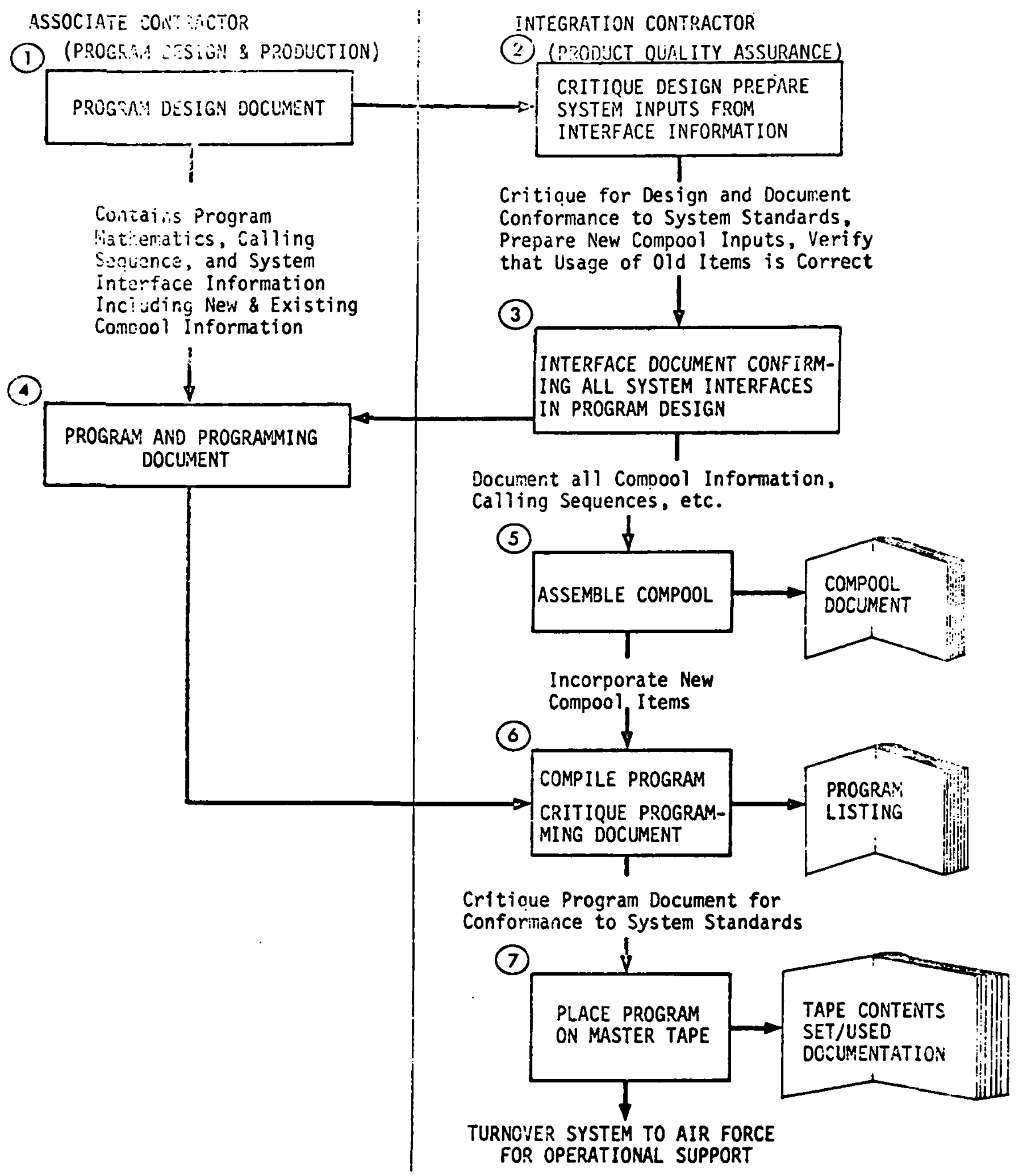

ASSOCIATE CONTRACTOR
1 (PROGRAM DESIGN & PRODUCTION)
PROGRAM DESIGN DOCUMENT
Contains Program Mathematics, Calling Sequence, and System Interface Information Including New & Existing Compool Information
4
PROGRAM AND PROGRAMMING DOCUMENT
INTEGRATION CONTRACTOR
2 (PRODUCT QUALITY ASSURANCE)
CRITIQUE DESIGN PREPARE SYSTEM INPUTS FROM INTERFACE INFORMATION
Critique for Design and Document Conformance to System Standards, Prepare New Compool Inputs, Verify that Usage of Old Items is Correct
3
INTERFACE DOCUMENT CONFIRMING ALL SYSTEM INTERFACES IN PROGRAM DESIGN
Document all Compool Information, Calling Sequences, etc.
5
ASSEMBLE COMPOOL
COMPOOL DOCUMENT
Incorporate New Compool Items
6
COMPILE PROGRAM
CRITIQUE PROGRAMMING DOCUMENT
PROGRAM LISTING
Critique Program Document for Conformance to System Standards
7
PLACE PROGRAM ON MASTER TAPE
TAPE CONTENTS SET/USED DOCUMENTATION
TURNOVER SYSTEM TO AIR FORCE FOR OPERATIONAL SUPPORT

Figure 1.–Product flow through AFSCF configuration management milestones.

(2) The configuration management scheme, although not specifically a part of the software system, is so basic for insuring product quality that any discussion of documentation would be incomplete without a description of it. How production and quality assurance functions check and balance each other, and how new products flow through this controlled scheme will be discussed.

(3) The computer-produced documents, when combined with manually prepared portions of the design and programming documents, provide high-quality, total-system documentation. When and how these documents are generated and the information that they contain will be discussed.

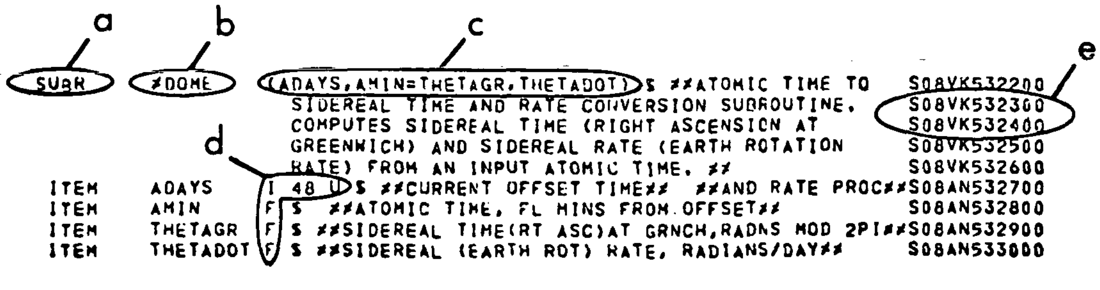
DATA-BASE DEFINITION

There are five basic elements in the AFSCF orbital prediction and command system that will be referenced throughout this paper. These are COMPOOL, the library tape, the master tape, configuration management, and the system programs. Of these, COMPOOL is the most fundamental, for it provides, in one centralized location, the basic definitions, references, and formats used by all the programs in the system.

Specifically, COMPOOL consists of names and calling sequence descriptions for all AFSCF system programs, along with descriptions of all system intercommunication data. Initially, COMPOOL takes the form of a punched card deck containing all the necessary descriptive information in the prescribed format. A special COMPOOL assembly program then processes these cards and compiles the data into tables that are an efficient input for the system compiler. During the COMPOOL assembly, a tape is produced that contains all the information on the input cards. This tape forms the basis for later COMPOOL updates, and it also serves as an input to a program that generates all the COMPOOL documentation. Data in the COMPOOL are organized according to the following hierarchy: blocks, tables (or arrays), and items, blocks being the gross data sets and tables and items being subsets of the blocks. As has been mentioned, COMPOOL also contains program descriptions and calling sequences.

Figure 2 is an example of the program calling sequence that is a part of the COMPOOL output. The explanation for labels *a* to *e* are as follows:

a: The notation SUBR indicates that the data that follow are for a subroutine (computer program). All programs are identified by SUBR.



```
SUBR      ≠DOME      (ADAYS,AMIN=THETAGR,THETADOT)$ ≠≠ATOMIC TIME TO    S08VK532200
                      SIDEREAL TIME AND RATE CONVERSION SUBROUTINE.      S08VK532300
                      COMPUTES SIDEREAL TIME (RIGHT ASCENSION AT         S08VK532400
                      GREENWICH) AND SIDEREAL RATE (EARTH ROTATION       S08VK532500
                      RATE) FROM AN INPUT ATOMIC TIME. ≠≠                S08VK532600
ITEM      ADAYS      I 48 U$ ≠≠CURRENT OFFSET TIME≠≠  ≠≠AND RATE PROC≠≠S08AN532700
ITEM      AMIN       F $  ≠≠ATOMIC TIME, FL MINS FROM OFFSET≠≠           S08AN532800
ITEM      THETAGR    F $ ≠≠SIDEREAL TIME(RT ASC)AT GRNCH,RADNS MOD 2PI≠≠S08AN532900
ITEM      THETADOT   F $ ≠≠SIDEREAL (EARTH ROT) RATE, RADIANS/DAY≠≠      S08AN533000
```

Figure 2.—Program calling sequence (excerpt from COMPOOL document).

b: DOME represents the name of the program (the ≠ symbol precedes all COMPOOL names).

c: These are input and output parameters. All parameters to the right of the ≠ sign are output by the program.

d: These are format definitions for the input and output parameters. The symbols I, 48, U, and F designate such information as floating, integer, signed, and the number of bits occupied by this item. The general data structures are defined in a system standards and conventions manual.

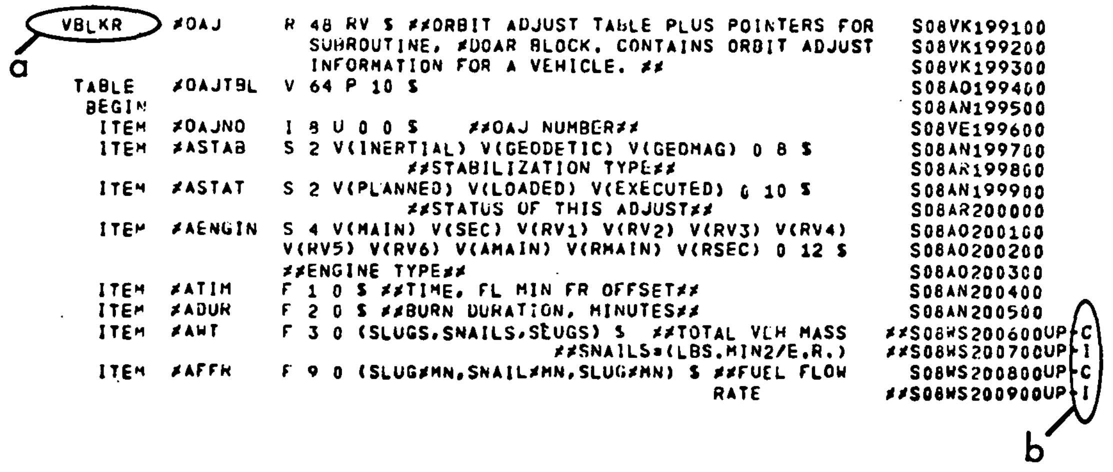
e: These are sequence numbers that permit updating the data definitions on tape without having to work with the entire card deck.

The definitions, enclosed within the ≠ ≠ symbols, are not required by the software system but are supplied for almost all entries. This is, of course, one of the capabilities that makes the COMPOOL document so valuable to users of the system.

Figure 3 shows a sample of the program interface data. The explanations for labels *a* and *b* are as follows:

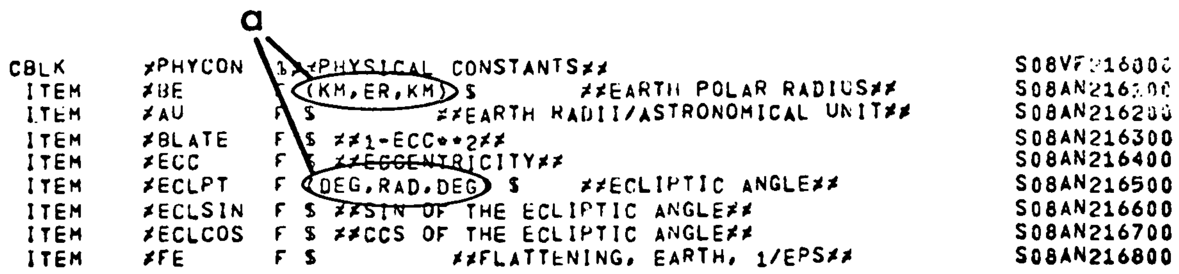
a: The notation BLK indicates a data block, V indicates the type of block (variable length), and R means that the data are automatically retrieved and stored at set intervals during a run to allow for restarts and subsequent runs. All entries are either SUBR or BLK entries, although the type of block and the retrieval information may vary. The balance of the information describes the data and format of the BLK in a form that is similar to the calling-sequence item descriptions in figure 2. All of the format information is defined in the system standards and conventions manuals.

b: The symbols C and I to the right of the sequence numbers indicate changes to and insertions in, respectively, the previous version of COMPOOL, which was used to produce this one.



```
VBLKR     ≠OAJ       R 48 RV $ ≠≠ORBIT ADJUST TABLE PLUS POINTERS FOR   S08VK199100
                       SUBROUTINE, ≠DOAR BLOCK. CONTAINS ORBIT ADJUST   S08VK199200
                       INFORMATION FOR A VEHICLE. ≠≠                    S08VK199300
  TABLE   ≠OAJTBL    V 64 P 10 $                                        S08AO199400
   BEGIN                                                                S08AN199500
    ITEM  ≠OAJNO     I 8 U 0 0 $     ≠≠OAJ NUMBER≠≠                     S08VE199600
    ITEM  ≠ASTAB     S 2 V(INERTIAL) V(GEODETIC) V(GEOMAG) 0 8 $        S08AN199700
                             ≠≠STABILIZATION TYPE≠≠                     S08AR199800
    ITEM  ≠ASTAT     S 2 V(PLANNED) V(LOADED) V(EXECUTED) 0 10 $        S08AN199900
                             ≠≠STATUS OF THIS ADJUST≠≠                  S08AR200000
    ITEM  ≠AENGIN    S 4 V(MAIN) V(SEC) V(RV1) V(RV2) V(RV3) V(RV4)     S08AO200100
                     V(RV5) V(RV6) V(AMAIN) V(RMAIN) V(RSEC) 0 12 $     S08AO200200
                     ≠≠ENGINE TYPE≠≠                                    S08AO200300
    ITEM  ≠ATIM      F 1 0 $ ≠≠TIME, FL MIN FR OFFSET≠≠                 S08AN200400
    ITEM  ≠ADUR      F 2 0 $ ≠≠BURN DURATION, MINUTES≠≠                 S08AN200500
    ITEM  ≠AWT       F 3 0 (SLUGS,SNAILS,SLUGS) $  ≠≠TOTAL VCH MASS   ≠≠S08WS200600UP C
                                         ≠≠SNAILS=(LBS,MIN2/E.R.)   ≠≠S08WS200700UP I
    ITEM  ≠AFFR      F 9 0 (SLUG≠MN,SNAIL≠MN,SLUG≠MN) $ ≠≠FUEL FLOW     S08WS200800UP C
                                                      RATE          ≠≠S08WS200900UP I
```

Figure 3.–Program interface data (excerpt from COMPOOL document).



```
CBLK     ≠PHYCON   1 ≠≠PHYSICAL CONSTANTS≠≠                        S08VF216000
 ITEM    ≠BE       F (KM,ER,KM) S       ≠≠EARTH POLAR RADIUS≠≠     S08AN216100
 ITEM    ≠AU       F S              ≠≠EARTH RADII/ASTRONOMICAL UNIT≠≠  S08AN216200
 ITEM    ≠BLATE    F S ≠≠1-ECC**2≠≠                                S08AN216300
 ITEM    ≠ECC      F S ≠≠ECCENTRICITY≠≠                            S08AN216400
 ITEM    ≠ECLPT    F (DEG,RAD,DEG) S    ≠≠ECLIPTIC ANGLE≠≠         S08AN216500
 ITEM    ≠ECLSIN   F S ≠≠SIN OF THE ECLIPTIC ANGLE≠≠               S08AN216600
 ITEM    ≠ECLCOS   F S ≠≠CCS OF THE ECLIPTIC ANGLE≠≠               S08AN216700
 ITEM    ≠FE       F S          ≠≠FLATTENING, EARTH, 1/EPS≠≠       S08AN216800
```

Figure 4.–System constants information (excerpt from COMPOOL document).

Figure 4 shows an example of the system constants information (CBLK), which is similar in form to the VBLK interface data (fig. 3). The C means constant: If a program references an item in this block, it is automatically loaded by the system executive each time the program is operated. The set of three units in the item definition (examples noted by *a*) indicates input, internal, and output units, respectively, for this item. These are nominal and can be overridden on the program request cards.

It should be recognized that COMPOOL does not contain any system data itself. For instance, the PHYCON block in figure 4 does not actually give the values of the constant, but only the description. The constants will ultimately be placed in a block by the input of their values and names (e.g., 10.25 and OMEGA, for Earth rotation rate) into a utility program that uses the assembled COMPOOL to determine the proper block, format, and units for the items. The PHYCON block would then be stored on tape for later use by the operational programs.

The operational programs will not use COMPOOL once they are compiled, however. When a program is first written, COMPOOL items are referenced by name. When the compiler encounters one of these COMPOOL data (or program) references during a compilation, the COMPOOL assembly tables are consulted, and the proper machine code is inserted for manipulation of the data. No computer program in the system, then, contains any interface data definitions, and the program does not directly reference any COMPOOL definitions, it references only the data names.

A number of controls insure the integrity of the COMPOOL document. First, the overall design of the system rules against the inclusion of programs in the system that are not included in COMPOOL. For instance, both the executive and compiler print error messages if a program is used that does not exist in COMPOOL. Second, interface data items cannot be used at all if they are not in COMPOOL before compilation. (However, test and development COMPOOL's may be used before formal submittal of the program for inclusion in the system.) Third, the configuration management system, discussed in the next section, insures that all programs and data in the system are properly entered in COMPOOL. Consequently, the COMPOOL document is a current, thorough, and accurate guide to the descriptions and calling sequences for all system programs and the descriptions and formats of all interface data. COMPOOL and the COMPOOL document simplify program access, use, and maintenance in several ways:

The document is always current because it is produced automatically with each COMPOOL update; it does not depend on the update being done manually. (The actual document production is discussed in the section entitled "Automated Documentation.")

Users can quickly ascertain whether a system program already exists to meet a certain requirement and, if it does and can be called by another program, what calling parameters are necessary. The document excerpts discussed in this section indicate the various COMPOOL entries and the amount of information that is available to the users.

Availability of the COMPOOL document helps maintenance and development personnel decipher portions of programs that reference COMPOOL data. Also, well-documented and accessible data definitions aid considerably in the production of new and modified programs.

The ability to reference data by name instead of having to include actual values in a program eliminates many mistakes and insures consistency in program results. This, in turn, aids in keeping the program and documentation in close harmony.

Constant data can be changed in one place, and all program references are automatically made to the new value. This is because the data can only be called by name in the program. The compiler converts this name to a location reference, and all references are made to this single location. The actual value of the constant need never be stated in the program, it need only be given in the library tape document (described in the section entitled "Automated Documentation"). The automated documentation of constants in a single location, then, is made possible by COMPOOL.

COMPOOL table or format changes require the recompiling of the programs that reference the altered items. However, programmers are not burdened with manual modification of data and definitions in the several programs that may require recompilation. All such modifications are accomplished automatically by the interaction between the COMPOOL and the compiler. Thus programmer errors are eliminated.

The next section will show how the configuration management scheme for the AFSCF software interacts with the software to keep the system user's information sources current and accurate.

CONFIGURATION MANAGEMENT

The COMPOOL document and COMPOOL itself are extremely valuable aids in the standardization and use of the software system. They offer no guarantee, however, that programs will conform to all system standards or that other program documentation will be adequate. To insure that deliverable products are complete, correct, and consistent, checks on product development, adequacy, and compliance with schedules and standards and balances among the skills, methods, and resources available to do the job effectively and efficiently are needed. In AFSCF operations, these checks and balances are provided by a configuration management system.

Configuration management refers to the planning, direction, and control of all factors affecting the state of the system. Some examples of configuration management tasks in the AFSCF system are new program scheduling, review of inputs and determination of COMPOOL content, determination of programs to be included on the master tape, and quality assurance through the critique and testing of new programs and documents. The balance of this section explains who fulfills the configuration management role in AFSCF, how products flow through the check and balance system, and what some of the specific controls are that provide for quality documentation.

Agencies and Responsibilities

The customer, while having ultimate authority over the system configuration, lacks adequate manpower to directly manage the overall effort. At the same time, the program production, or associate, contractor is too involved with costs and schedules to provide the necessary unbiased support, particularly in the area of quality assurance. Consequently, a third party, known as the integration contractor, is employed to support the customer's configuration management efforts, particularly in the quality assurance and product acceptance areas.

For the past 9 years, SDC has fulfilled this role for the AFSCF software system. The specific tasks performed in this integration role are the detection, definition, and resolution of interface problems (between contractors, programs, and hardware); checking individual programs for conformance to design specifications, standards, and conventions; integration of the programs into a complete computer program subsystem; and validation of the subsystem to insure that all elements of the package are operational and compatible.

Product Development

The checks and balances, then, are between the integration contractor and the associate contractor. Figure 1 delineates the roles of these two contractors as new computer programs pass through the AFSCF product development cycle. The configuration management scheme, based on U.S. Air Force Exhibit 61-47B, was designed by SDC specifically for AFSCF. The following are some aspects of this scheme that aid in securing quality documentation and a quality software system.

The mathematical development for a program is presented in the program design document (step 1, fig. 1). The integration contractor has time to evaluate and digest this information (steps 2 and 3, fig. 1) before the actual program release. The integration contractor also reviews the design document to insure that provision has been made to place new interface data elements in COMPOOL, to properly use the existing COMPOOL information, and, in particular, to insure that no data are imbedded in the program that should be COMPOOL definitions.

When the integration contractor receives the program (step 4, fig. 1), it is manually compared with the programming and design documents. The two products must conform before the program is accepted. This conformance is also required for design logic or any other portion of the design document that could cause problems in later development or analysis work. Programmer analysts rather than computer programs are used to perform these checks. This work is done manually for the simple reason that although the automatic flowchart program can track any arithmetic and logic, no automated method exists that will deduce mathematical derivations or the logical bases of certain techniques from the program (computer code) itself.

The conformance between programs and the COMPOOL items they use is imposed by the system itself. The COMPOOL inputs are prepared from the design document (steps 2 and 5, fig. 1), and the program is then compiled with this COMPOOL. This procedure automatically forces conformance between the program and COMPOOL and also guarantees that the program matches the design document (in the COMPOOL area), the design document being the source of COMPOOL inputs. It follows, then, that COMPOOL, the COMPOOL document, the program, and the design document are all consistent.

Quality Control Measures

The system standards and conventions are enforced throughout the review of the documentation and the final program acceptance. There is a single standards and conventions document for the AFSCF orbital prediction and command system. Briefly, some of the areas covered by this document are–

(1) COMPOOL inputs–conventions, format standards, necessary description information, block sizes, etc.
(2) Data cards–format standards, use of special columns, error processing conventions, etc.
(3) Documentation–format and content standards, program calling sequence, illustration conventions, etc.
(4) Executive interface–illegal instruction standards, input/output (I/O) usage, program size requirements, successor call and nesting standards, compiler usage, etc.
(5) I/O usage–choice of units, access methods, record sizes, internal tape label information, lines per page, and heading information required by system
(6) Messages/error detection–requirements for error messages, error message output devices, system error messages, etc.

This list is only a sample of the areas covered; however, it indicates the extent to which the system is governed by standards and conventions. How these restrictions and guidelines aid the documentation task can be illustrated with an example of input parameters for time. Suppose that the time "1330 hours, 59.2 seconds, 3 June 1970" was an input parameter on a data card. The possible variations in input format for these data are almost innumerable. Some of the possibilities are:

3 6 70 13 30 59.2,

3 6 1970 1330 59 2,

3 JUNE 70 13 30 59.2,

JUNE 3 1970 1330 59 2

Definition of a standard input format for this example accomplishes several things. The program documentation is easy to read, the interpretation of each of the parameters comprising the time is unnecessary because there is only one format, and typographical errors are of little concern because a single format removes all ambiguity or misunderstanding. Users can quickly format data cards without fear that a particular program does things a little differently. Finally, a common system program can be used to convert the time input to different reference bases, thus relieving the programmer of a task that would have to be tested, with both the test and program logic requiring documentation. The development and enforcement of system standards, then, can be a great step forward in the solution of documentation problems. The application of standards, incidentally, is generally enforced by the program system as well as by the configuration management scheme. This is because many of the standards have to do with interfaces among existing system programs; violations in these circumstances generally result in error messages and unsuccessful computer runs.

When a new or modified version of a program is delivered to the integration contractor, that program must have a unique identification to differentiate it from all other versions of the same program. That identification is known as the "mod." The mod identifier is compiled with the program and is automatically transmitted whenever the program is loaded onto the master tape. Because the configuration management scheme "freezes" the program upon formal submittal to the system (by directing that the programs be stored on specially controlled tapes), all listings bearing the same mod number are guaranteed to be identical and are an accurate reflection of the program bearing that mod on the master tape. Similarly, the master tape has an identification that is printed out on all computer runs, logs, etc. This identification changes whenever a change occurs in the master tape.

Automated documentation is produced and maintained for each version of the master tape. This documentation shows the exact contents, including program mods, of the tape. Because each configuration of master tape and program is unique and because each is covered by documentation, all guesswork concerning the identification of a configuration undergoing maintenance or troubleshooting is removed. The high degree of interdependence among the system programs in the AFSCF system makes knowledge of the correct configuration particularly essential because different versions of the master tape may be current at the same time in support of different projects.

Many phases of configuration management, such as change control and scheduling, are not discussed here, not because they are unimportant to documentation, in fact, they are quite important, but because their importance is somewhat less tangible and more difficult to explain. The aspects that are presented, however, show how a strong quality assurance endeavor, backed by software expertise and well-defined standards and procedures, can greatly improve the quality of system documentation.

AUTOMATED DOCUMENTATION

The AFSCF automated documentation touches on all elements of the system: COMPOOL, master tape, library tape, configuration management status information, and all the computer programs. Figure 5 shows these elements and the inputs and outputs that comprise the automated documentation scheme. A major factor in the functioning of the automated documentation is the centralization of data: the master tape contains all the programs and configuration information, the library tape and COMPOOL contain data normally found only in the individual computer programs, and the status information provides complete details for the three areas of status shown in figure 5. Along with the centralization of data, of course, the accuracy of the documentation depends on the use of the strict configuration management methods described in the previous section.

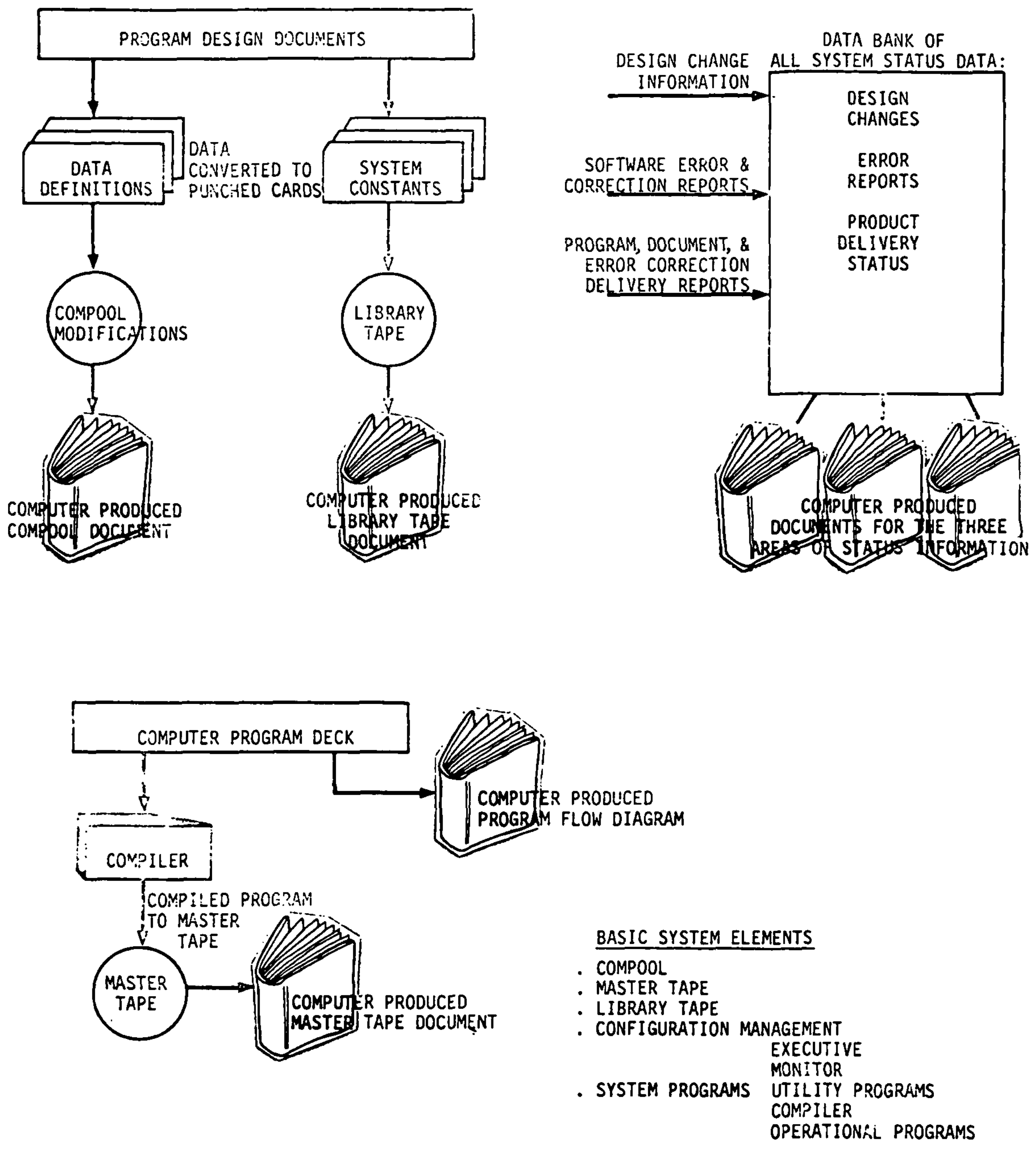

PROGRAM DESIGN DOCUMENTS
DATA DEFINITIONS
DATA CONVERTED TO PUNCHED CARDS
SYSTEM CONSTANTS
COMPOOL MODIFICATIONS
LIBRARY TAPE
COMPUTER PRODUCED COMPOOL DOCUMENT
COMPUTER PRODUCED LIBRARY TAPE DOCUMENT
DESIGN CHANGE INFORMATION
DATA BANK OF ALL SYSTEM STATUS DATA:
DESIGN CHANGES
ERROR REPORTS
PRODUCT DELIVERY STATUS
SOFTWARE ERROR & CORRECTION REPORTS
PROGRAM, DOCUMENT, & ERROR CORRECTION DELIVERY REPORTS
COMPUTER PRODUCED DOCUMENTS FOR THE THREE AREAS OF STATUS INFORMATION
COMPUTER PROGRAM DECK
COMPUTER PRODUCED PROGRAM FLOW DIAGRAM
COMPILER
COMPILED PROGRAM TO MASTER TAPE
MASTER TAPE
COMPUTER PRODUCED MASTER TAPE DOCUMENT
BASIC SYSTEM ELEMENTS
. COMPOOL
. MASTER TAPE
. LIBRARY TAPE
. CONFIGURATION MANAGEMENT
. SYSTEM PROGRAMS
EXECUTIVE
MONITOR
UTILITY PROGRAMS
COMPILER
OPERATIONAL PROGRAMS

Figure 5.–Automated documentation scheme.

The documents depicted in figure 5 are not necessarily useful as separate entities but are quite useful when taken together as a system document. For instance, the library tape document provides the names and values of system constants, but the COMPOOL document provides the actual description of the data. The master tape documentation is supported by the configuration management data base, which describes significant features of the new program mods and the status of all errors and corrections affecting the tape. COMPOOL and library tape documents and the automatic flowcharts complement the manually produced mathematic and logic documents of individual programs. The balance of this section describes some of these computer-produced documents.

COMPOOL Documentation

The actual production of the COMPOOL document is governed by a program that accepts the list tape (effectively a card image tape), places heading information on the page, and then prints approximately 50 lines of card images. The heading information is requested by a program control card and consists of date, document name and number, starting page number, etc. This program is general purpose; it will accept all AFSCF system list tapes as input and will allow sources of input to be alternated so input cards can be used to annotate information coming from the tape. Figure 6 is an example of the heading output of this program.

Library Tape Documentation

In most systems, the library is the respository for programs, but, in the AFSCF system, the library tape contains the system constant information. The basic form of the input to the library tape program is punched cards, with a list tape being produced along with the library tape itself. Both tapes are processed by the same program that produces the COMPOOL document. Figure 7 is a sample of the library tape document. All the element names are COMPOOL entries, and the sample shown here conforms with the COMPOOL entries in figure 4.

Master Tape

The master tape contains all the computer programs in the system. During the compilation of a program, the compiler sets up tables containing information on all COMPOOL references (both programs and data). These tables are then transferred to the master tape when the program is loaded onto the tape. The program that documents the master tape references these tables, and, along with a log of the programs on the tape, it can produce complete set/use references for all programs and data items. This output can then be placed in the individual program documents (see fig. 1) for subsequent use by the system users and maintenance personnel. Samples of the master tape documentation are shown in figure 8.

System Status Documentation

Figure 5 shows the three areas of status information that are handled by computer documentation. The types of data in the data bank include

2 of 3

```
                                                      SYSTEM DEVELOPMENT CORPORATION
15 JUNE 1970                        44                       TM-(L)-4164/314/00
```

Figure 6.–Page heading produced by automated documentation program.

```
ELEMENT ≠PHYCON/G                          $                                      ADQ   062700
        ≠BE           0.6356775E+4         $                                      ADO   062800
        ≠AU           0.23454710E+5        $                                      ADG   062900
        ≠BLATE        0.9933054E+0         $                                      ADG   063000
        ≠ECC          0.81820E-1           $                                      ADG   063100
        ≠ECLPT RAD    0.409206212          $                                      ADG   063200
        ≠ECLSIN       0.397881208          $                                      ADG   063300
        ≠ECLCOS       0.917436945          $                                      ADG   063400
        ≠FE           0.298250E+3          $                                      ADG   063500
```

Figure 7.–Values of system constants (excerpt from library tape document).

(1) Design changes–descriptions of all proposed design changes, identified by control numbers. Status conditions include accepted for future implementation, rejected, and pending action.

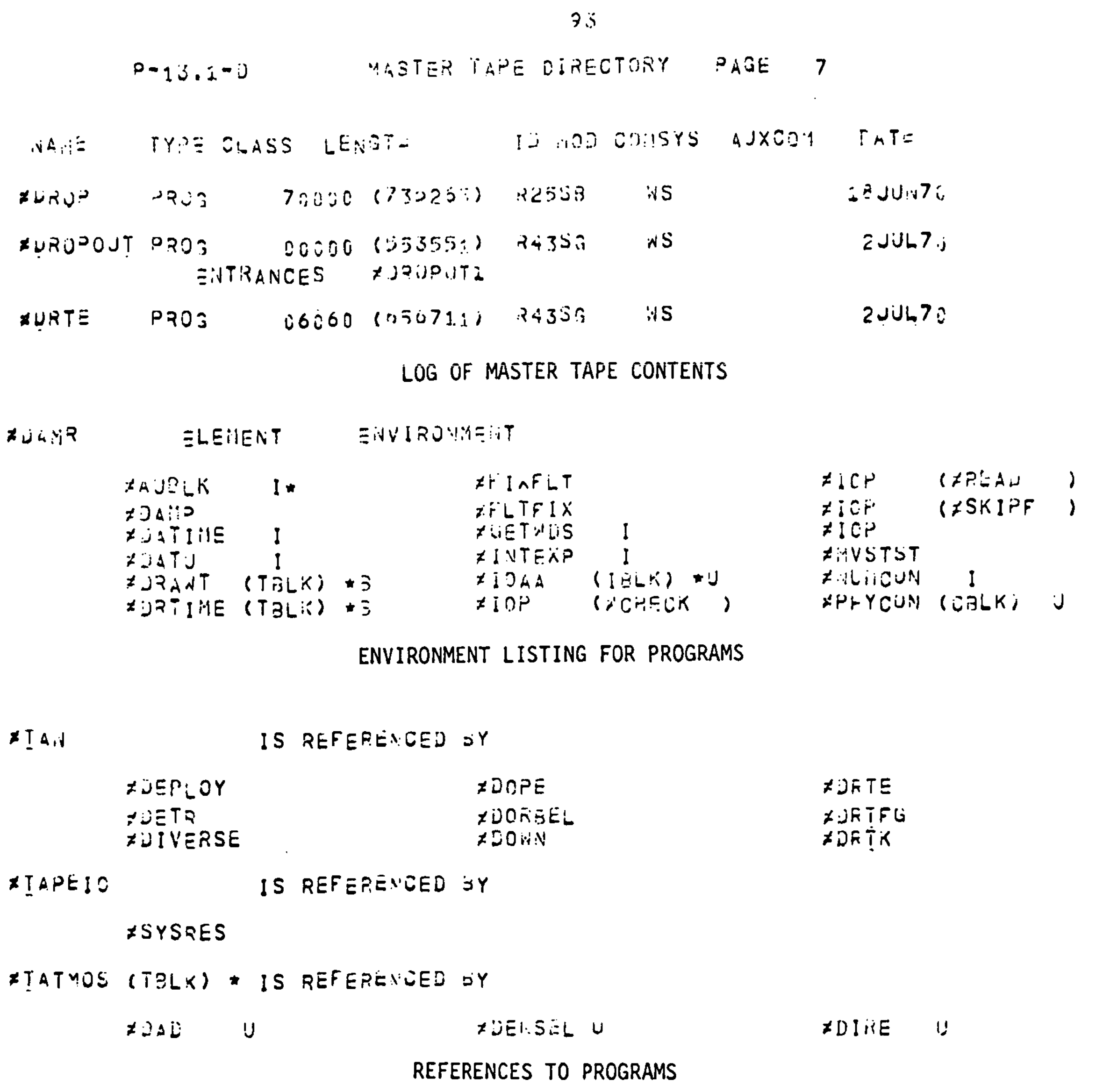
(2) Error reports–descriptions of all reported errors in the AFSCF software system, identified by control number and program or by document number, priority, responsible agency, etc.

(3) Product delivery status–description of all program, document, and program corrector deliveries, identified by control number, delivering agency, applicable error report numbers for correctors, etc.

Special report-generating programs employ these data for regular status reports and user information documents when new master tapes are released to the customer. These programs can produce reports with data sorted by control number, status (open or closed problems, scheduled or rejected design changes, etc.), program names, priority of problem, etc. The programs also compile status summaries for the three areas of information. Sample outputs are shown in figure 9.

Flowchart Documentation

The individuality of programmers affects flowcharts more than any other portion of a program document. Although symbols can be standardized, the level of detail is very difficult to regulate, flowchart accuracy is almost as difficult to monitor as program accuracy, and the flow invariably reflects what the programmer wants the program to do and not necessarily what it does. An automatic (computer) flowcharter overcomes all these problems; it is consistent in level of detail and reflects exactly what the program does. Furthermore, it is available as soon as the program is available, which is much more timely than the typical manually produced diagram.



```
                                   93

     P-13.1-D          MASTER TAPE DIRECTORY    PAGE    7

 NAME      TYPE CLASS  LENGTH          ID MOD CONSYS  AJXCOM   DATE

*DROP      PROG        70000 (732257)  R25SB    WS              18JUN70

*DROPOUT  PROG        00000 (553551)  R43SG    WS               2JUL70
                ENTRANCES     *DROPUT1

*DRTE      PROG        06060 (056711)  R43SG    WS               2JUL70

                           LOG OF MASTER TAPE CONTENTS

*DAMR           ELEMENT       ENVIRONMENT

          *ADBLK    I*             *FIXFLT                        *ICP     (*READ    )
          *DAMP                    *FLTFIX                        *ICP     (*SKIPF   )
          *DATIME   I              *GETMDS    I                   *ICP
          *DATU     I              *INTEXP    I                   *MVSTST
          *DRAWT  (TBLK) *S        *IDAA     (IBLK) *U            *NUMCON    I
          *DRTIME (TBLK) *S        *IOP      (*CHECK  )           *PHYCON (CBLK)   U

                         ENVIRONMENT LISTING FOR PROGRAMS


*IAN                    IS REFERENCED BY

          *DEPLOY                  *DOPE                          *DRTE
          *DETR                    *DORSEL                        *DRTFG
          *DIVERSE                 *DOWN                          *DRTK

*IAPEIO                 IS REFERENCED BY

          *SYSRES

*IATMOS (TBLK) * IS REFERENCED BY

          *DAD      U              *DENSEL U                      *DIRE    U
                           REFERENCES TO PROGRAMS
```

Figure 8.–Excerpts from computer-produced master tape documentation.

The AFSCF system has an automatic flowchart program (FLOW) that is currently applied to a majority of the system program documents.

FLOW is designed specifically to work with the system compiler (JOVIAL) and to analyze JOVIAL language statements. It will recognize direct, or machine, code, but it merely sets these off in a box on the diagram.

FLOW accepts prestored tape or card decks (of the object program), and a printer then outputs the flowchart. A more ideal output device for a flowcharter is a plotter, but there

```
                                        **** SUMMARY OF DRF#S - CLOSED ****

DRF    PROG.NAME MID-MOD PRI-   STATUS  SUBSYS. ORIG. DRF DATE  S/U DATE  HOW     DISPO-      PROG.NAME MID-MOD  MTM   **CHANGED DATA
NUMB.     DRF      DRF   ORITY                  CO.                       CLOSED  SITION        S/U       S/U    NUMB. ***NEW ENTRY

Z0659  #SPRP     P61VC   LOW    CLOSED    E/U    SDC  11/04/68  07/20/70  NEW    P13.1 PEND  #SDARE     P64NG   T00     *
Z1291  #CARDIO   P74WA   MED    CLOSED    E/U    SDC  08/07/69  07/20/70  NEW    P13.1 PEND  #CARDIO    P74WB   T00     *
Z1448  #TAPEIO   P73WD   LOW    CLOSED    E/U    SDC  09/22/69  07/16/70  NEW    P13.1 PEND  #TAPEIO    P73WE   T00     *
Z1582  SYMON             LOW    CLOSED    E/U    SDC  11/13/69  07/20/70  NEW    P13.1 PEND  #SYSRES    O10WE   T00     *
Z1593  #DEPLOY   T210C   LOW    CLOSED    OPS    SDC  11/20/69  07/14/70  NEW    P13.1 PEND  #DEPLOY    T21SE   T00
Z1594  #DEPLOY   T210C   LOW    CLOSED    OPS    SDC  11/20/69  07/14/70  NEW    P13.1 PEND  #DEPLOY    T21SE   T09
```

SUMMARY INFORMATION

```
*** DRF#S ON OPERATIONAL SUBSYSTEM CLOSED SINCE 8/18/70 =    27 ***

HIGH PRIORITY =     1
  Z2280

MEDIUM PRIORITY =    13
  Z2023  Z2050  Z2118  Z2138  Z2148  Z2147  Z2163  Z2166  Z2172
  Z2182  Z2235  Z2238  Z2291

LOW PRIORITY =      13
  Z1889  Z2013  Z2022  Z2148  Z2159  Z2171  Z2234  Z2237  Z2253
  Z2267  Z2271  Z2292  Z2312
```

SUMMARY INFORMATION

```
#DECOR          076  RD  E05                 Z1643          02/04/70
    OCTAL CORRECTORS TO #DECOR TO CORRECT THIS DRF. THESE OCTALS SUBMITTED
    12-10-69 WILL CORRECT THE PROBLEM ONLY AFTER A NEW COMPOOL IS BUILT TO
    INCREASE THE NUMBER OF BITS IN #PASREV. A CCR HAS BEEN SUBMITTED.
    S/U 02/04/70 DRF Z1643 #DECOR   E05 076 RD MTM SUBMITTED
        EXPLANATION ACCEPTED. CLOSURE WILL BE POSTPONED UNTIL INTFGRATION
        OF CCR.
```

STATUS FOR SPECIFIC ERROR REPORT

Figure 9.–Excerpts from computer-produced system status reports.

are no plotters available in the AFSCF hardware inventory. Examples of the construction of various flowchart symbols made with printer characters are shown in figure 10.

FLOW can produce flowcharts at several levels of compression. The initial level is approximately one box per input statement. Starting at this level, a set of seven rules is applied to collapse the diagram. The amount it can be collapsed depends ultimately on the logic of the program. In any case, the prime value of this capability is that the amount of collapsing can be controlled at many interim points between the first and ultimate levels. Thus, flowcharts can be produced that are shorter than the initial level but are still at a level of detail that fully reflects the program logic.

FLOW produces the charts in a form that is suitable for inclusion in a document. The output is small enough that it is suitable for the conventional 8½- by 11-in. page, and all heading information, such as document number, page numbers, and date, are provided.

The program offers options for processing only portions of a program; complete statements in JOVIAL can be output rather than their translations, etc.

The basic advantage gained by the use of FLOW (or any flowcharting system) is that it permits timely and accurate logic flows of the program to be made. In the AFSCF software system there is the additional advantage of being able to cross-reference FLOW output

```
                         127

      .0809.              .*.
                        .*   *.
                      .*  IS   *.
                    .* CHN($I$)  *.YES             ----
                   *.GR 7 <TRUE>> .*>>>>>>>>>>>>(0813)
                    *.           .*                ----
                      *.       .*                 OCTERR
                        *.   .*
                          * NO
                          *
      .0810.              *    2 ENTRIES
           ***************************
           * SET DCON; INCREMENT I   *
           * BY 1.                   *
           ***************************
                          *
                         .*.
                       .*   *.
                     .*  IS   *.
----            YES .* I LQ LCOL *. NO                  ----
(0806)<<<<<<<<<<<<<*.  <TRUE>>   .*>>>>>>>>>>>>(0814)
----                 *.         .*                      ----
                       *.     .*
                         *. .*
                          *


      .0813.   OCTERR     *    2 ENTRIES
           ***************************
           **ENTER PROCEDURE        **
           **FLDERR.                **
           ***************************
                          *
      .0814.              *    2 ENTRIES
           **************************
           * RETURN FROM PROCEDURE  *
           **************************


           **************************
           *   PROCEDURE FLDERR     *
           **************************
                          *
                          *
                          *
                          *
                          *
```

Figure 10.–Automatic flowcharter output.

with the library tape and the COMPOOL documentation. This provides the equivalent of an annotated flowchart with considerably more information content than is available from flowcharts of self-contained programs.

CONCLUSIONS AND RECOMMENDATIONS

The benefits and advantages of the AFSCF system, thorough, accurate, timely, and automated program documentation, are features that would be desired by the users of any system. The question of how to relate the design concepts in this paper to other systems can be considered with the following three facts concerting both this paper and the AFSCF system.

First, the purpose of this paper is not to show specifically how a system should or must be designed but rather to show what can be accomplished by integrating documentation into the basic design. It is unlikely that the same design would be the best approach in any other system, but certainly the principles of design, such as centralization of data, rigid control of the configuration, and program-imposed standards, are valid in other systems.

Second, the AFSCF system was designed under ideal circumstances in that the compiler, executive, monitor, and configuration management techniques were all part of the design effort. This is a tremendous advantage over having to develop a system around already existing compilers and executives, the most common approach to system design.

Finally, the cost of a potential error is so high that AFSCF invests very heavily in "insurance" procedures. The rigid configuration management controls described in this paper are a good example of that. Certainly, it is expensive to critique and review every document and intensively test every program brought into the system. Controlling the master tape, library tape, and COMPOOL is also expensive, but the cost of losing one satellite because of software errors makes the error "insurance" an excellent investment. There are some systems for which errors may be less costly and would not warrant the type of procedures that AFSCF employs.

In spite of the specialized aspects of the AFSCF system, the design concepts are valid for any system. For instance, the centralized data approach is highly desirable, but it is not necessarily practical to implement this approach in an already existing system. However, a data base containing all the desired information for documents can be compiled by requiring that information be incorporated in each program in the form of "pseudodata" (information not required by the compiler, such as comments). These pseudodata must not affect operation of the system compiler. A preprocessing program could then be used to extract this information for documentation. Further, setting standards for the pseudodata and incorporating legality checks on these standards in the preprocessor would help one secure information with a minimum of manual intervention.

Standards and conventions are of value even if they are not rigidly enforced; this is particularly true for documentation content and input formats. The mere fact that programs are produced under standards can provide the user with a good deal of information, even without the use of any specific program documentation. Furthermore, programmers will generally conform to reasonable standards and conventions if they are available. The real interest in automated documentation is not the production of documents; it is making information available, and standardization can help make information available with fewer documents.

Finally, whether the system is large or small, designed from scratch or only added, the key point is to plan the documentation and not let it just happen. The planned approach, along with some application of the principles presented here, will guarantee better quality documentation for any software system.

DISCUSSION

MEMBER OF THE AUDIENCE: Does the master tape documentation consist of the text or is it a set of module text descriptions?

WOLF: Basically, it is an index, containing program names, dates of loading, number of cells used, etc. One can go from the master tape documentation to the COMPOOL document, which contains the text. The several documents discussed, taken together, constitute the system document.

MEMBER OF THE AUDIENCE: How often do you produce a master tape?

WOLF: At the present time, one is produced every 2 months, although the need for documentation in certain circumstances will occasionally shorten this period to a few days.

N73-19213

SYNTAX-DIRECTED DOCUMENTATION FOR PL360*

Dr. Harlan D. Mills
IBM

PL360, due to the efforts of Niklaus Wirth (ref. 1), is a phrase-structured programming language which provides the facilities of a symbolic machine language for the IBM 360 computers. It is defined by a recursive syntax and is implemented by a syntax-directed compiler consisting of a precedence syntax analyzer and a set of interpretation rules, as discussed by Wirth and Weber in reference 2.

Syntax-directed documentation refers to an automatic process which acquires programming documentation through the syntactical analysis of a program, followed by the interrogation of the originating programmer. This documentation can be dispensed through reports or file query replies when other programmers later need to know the program structure and its details.

The interrogation of an originating programmer consists of a relisting of the program text, with certain syntactic entities, which are classified as documentation units, set off typographically in lines and labeled with an ordinal coordinate system and a sequence of questions about these documentation units. These questions are generated automatically by completing prestored skeleton questions with coordinates and/or programmer-generated identifiers. The programmer's responses to the questions are stored and indexed to these documentation units for retrieval.

A key principle in what follows is that the programming documentation process is managed solely on the basis of the syntax of programs. The semantics of the documentation, as embodied in programmer responses to interrogation, are not analyzed by the process except in mechanical ways such as keyword indexing. In this way, a programmer's responses are treated as "black messages" in the process, in analogy to the idea of a "black box." That is, a programmer's responses are requested, accepted, stored, and later retrieved with no semantic analysis of their contents.

SYNTACTIC PRELIMINARIES

We use the notation and definitions for PL360 in reference 1. In defining documentation units and lines, the following device is used. First, denote the grammar in reference 1 by G, which defines the language PL360, $L(G)$. This grammar G will be transformed finitely into a new grammar G^* such that

$$L(G^*) = L(G)$$

*Copyright © 1970, Association for Computing Machinery, Inc. Reprinted by permission.

```
 1   <BLOCK>            ::= <BLOCKBODY> END
 2   <CASE ST>          ::= <CASE SEQ> END
 3   <FUR ST>           ::= FOR <ASS STEP> <LIMIT> <DO> <STATEMENT*>
 4   <FUNC DECL7>
 5   <FUNC ID>
 6   <FUNC ST>          ::= <FUNC1> )
 7   <GOTO ST>          ::= GOTO <ID>
 8   <IF THEN ELSE ST> ::= IF <COND THEN> <TRUE PART> <STATEMENT*>
 9   <IF THEN ST>       ::= IF <COND THEN> <STATEMENT*>
10   <K REG ASS>
11   <NULL ST>
12   <PROC DECL>        ::= <PROC HD6> <STATEMENT*>
13   <PROC ID>
14   <PROGRAM>
15   <SEG DECL>         ::= <SEG HEAD> BASE <K REG>
16   <SYN DC2>
17   <T CELL ASS>       ::= <T CELL> := <K REG>
18   <T DECL7>
19   <WHILE ST>         ::= <WHILE> <COND DO> <STATEMENT*>
```

Figure 1.—Documentation units.

and such that G^* contains syntactic entities we want to classify as documentation units and use to define lines.

The basis for the transformation of G into G^* is a finite number of elementary steps as follows. If X is any finite sequence of tokens and/or syntactic entities which occurs as part of the right side of a production rule in a grammar G^k, and $\langle A\rangle$ is not a syntactic entity in G^k, we can define a new production $\langle A\rangle ::= X$ and substitute $\langle A\rangle$ for X in the right side of any rule we please in G^k, to get a grammar G^{k+1}. It is clear that $L(G^{k+1}) = L(G^k)$ by this construction. Then, we consider a sequence

$$G = G^0, G^1, \ldots, G^n = G^*$$

where n is the (finite) number of additional syntactic entities we want to be defined in G^* which are not in G.

We note that even though additional syntactic entities can easily be introduced in a grammar while retaining the identical language, the question of keeping it a precedence grammar (ref. 2) is a delicate matter. This general point is not pursued here. However, we use only transformations which label the entire right side of a rule; in this case the grammar obviously retains its precedence properties.

In what follows, the grammar G is augmented to G^* just to provide a basis for invoking additional interpretation rules which define documentation files and generate questions. It will also be apparent that the same device can be useful in extending syntax processing beyond documentation to questions of execution control and dynamic storage allocation in multiprogramming operating systems. For example, better use of core may arise if core is allocated to the machine code responding to syntactic entities such as "for statements" and "while statements" rather than simply arbitrary "pages" of machine code which may break up such natural units of execution.

DOCUMENTATION UNITS

We classify as a documentation unit any right-hand side of a rule which reduces to one of the following syntactic entities in reference 1:

⟨SIMPLE STATEMENT⟩
⟨STATEMENT⟩
⟨DECL⟩
⟨PROGRAM⟩

There are 19 such documentation units given in figure 1. If the right-hand side is already defined in G, it is used directly. Otherwise, a new syntactic entity is defined, with the understanding that G is augmented by each such definition, as described above.

In effect, this classification of documentation rules is a convenience for identifying productions whose recognition in an analysis corresponds to having additional interpretation rules that deal with documentation processing.

Given a PL360 program, we consider every realization of such documentation units, which can be structured on the basis of syntactic membership, as follows. A documentation unit is a member of a second documentation unit if its program text is a subset of the program text of the second. It is an immediate member if it is not a member of any third documentation unit, itself a member of the second.

The relation of immediate membership defines a nested structure of documentation units in a program, beginning with the program itself as the highest level documentation unit and continuing through "blocks," "compound statements," etc., to "single declarations" and "single statements" at the lowest levels. This nested structure can also be described as a rooted tree, with the program as the root, and other documentation units as remaining intermediate and endpoint nodes in the tree.

Notice any given statement or declaration may be included in the program text of many documentation units. In fact, every documentation unit is a member of the program and of every other documentation unit whose text contains it.

SYNTAX-DEFINED PROGRAM LISTINGS

Next, we consider the question of listing programs written in PL360 in a standard way for readability and referencing during programmer interrogation and later examination. When programmers make an informal effort to arrange their programs for readability, they typically start each documentation unit, as defined above, on a new line and use indentation to correspond in a general way with syntactical nesting in the program. We recognize that the problem is a subjective one, but we give a syntax-defined listing algorithm which is believed to satisfy the intuitive intentions observed in informal programming efforts.

For the purpose of typographical listing, we partition a PL360 program or procedure into a string of substrings. Each substring is to be a printed line, and the string of lines constitutes a listing of the program. Associated with each line are two numbers: one which specifies its order in the program or procedure, and one which corresponds to the indentation (or starting column) of the line. If a line exceeds the width of paper available, its continuation is further indented a standard amount.

The partition of a PL360 program or procedure into lines is defined by marking the starting text for each documentation unit, and each label, BEGIN, END, ELSE, and . (dot) symbol. The lines are numbered consecutively. The indentation number is the level of nesting of the documentation unit it begins, if any, based on syntactic membership as described above. The only lines not beginning a new documentation unit are BEGIN (in CASE statements), END, ELSE, and . (dot). In each case they are indented according to the level of the documentation unit which they help define. Labels are given the indentation level of the program or procedure being listed.

To refer to a line from outside a procedure, we qualify the line numbers with the procedure name. While the concept of program is defined in PL360, no provision is made for naming a program in the syntax.

```
<BLOCK>
    Q1  PURPOSE OF BLOCK (COORDINATES)?
    S1  BLOCK (COORDINATES) IS TO (RESPONSE).
<CASE ST>
    Q1  PURPOSE OF CASE STATEMENT (COORDINATES)?
    S1  CASE STATEMENT (COORDINATES) IS TO (RESPONSE).
    Q2  CASE SELECTED AT (COORDINATE)?
    S2  CASE SELECTED AT (COORDINATE) IS (RESPONSE).
<FOR ST>
    Q1  PURPOSE OF FOR STATEMENT (COORDINATES)?
    S1  FOR STATEMENT (COORDINATES) IS TO (RESPONSE).
    Q2  FOR CONDITION AT (COORDINATE)?
    S2  FOR CONDITION AT (COORDINATE) IS TO (RESPONSE).
<FUNC DECL?>
    Q1  FUNCTION OPERATION AT (COORDINATE)?
    S1  FUNCTION OPERATION AT (COORDINATE) IS TO (RESPONSE).
<FUNC ID>
    Q1  PURPOSE OF FUNCTION STATEMENT AT (COORDINATE)?
    S1  FUNCTION STATEMENT AT (COORDINATE) IS TO (RESPONSE).
<FUNC ST>
    Q1  PURPOSE OF FUNCTION STATEMENT AT (COORDINATE)?
    S1  FUNCTION STATEMENT AT (COORDINATE) IS TO (RESPONSE).
<GOTO ST>
    Q1  GO TO WHERE AT (COORDINATE)?
    S1  AT (COORDINATE) CONTROL GOES TO (RESPONSE).
<IF THEN ELSE ST>
    Q1  PURPOSE OF IF THEN ELSE STATEMENT (COORDINATES)?
    S1  IF THEN ELSE STATEMENT AT (COORDINATES) IS TO (RESPONSE).
    Q2  IF CONDITION AT (COORDINATE)?
    S2  IF CONDITION AT (COORDINATE) TESTS (RESPONSE).
<IF THEN ST>
    Q1  PURPOSE OF IF THEN STATEMENT (COORDINATES)?
    S1  IF THEN STATEMENT (COORDINATES) IS TO (RESPONSE).
    Q2  IF CONDITION AT (COORDINATE)?
    S2  IF CONDITION AT (COORDINATE) TESTS (RESPONSE).
<K REG ASS>
    Q1  VALUE OF (<ID>) AT (COORDINATE)?
    S1  VALUE OF (<ID>) AT (COORDINATE) IS (RESPONSE).
<NULL ST>
    Q1  PURPOSE OF NULL STATEMENT AT (COORDINATE)?
    S1  NULL STATEMENT AT (COORDINATE) IS TO (RESPONSE).
<PROC DECL>
    Q1  AUTHOR OF PROCEDURE (<ID>)?
    S1  AUTHOR OF PROCEDURE (<ID>) IS (RESPONSE).
    Q2  PURPOSE OF PROCEDURE?
    S2  PROCEDURE (<ID>) IS TO (RESPONSE).
    Q3  INITIAL DATA?
    S3  INITIAL DATA OF PROCEDURE (<ID>) IS (RESPONSES).
    Q4  PROCESSING LOGIC?
    S4  PROCESSING LOGIC OF PROCEDURE (<ID>) IS TO (RESPONSE).
    Q5  FINAL DATA?
    S5  FINAL DATA OF PROCEDURE (<ID>) IS (RESPONSE).
    Q6  REFERENCES?
    S6  REFERENCES FOR PROCEDURE (<ID>) ARE (RESPONSE).
<PROC ID>
    Q1  PURPOSE OF PROCEDURE STATEMENT AT (COORDINATE)?
    S1  PROCEDURE (<PROC ID>) AT (COORDINATE) IS TO (RESPONSE).
<PROGRAM>
    Q1  AUTHOR OF PROGRAM (<ID>)?
    S1  AUTHOR OF PROGRAM (<ID>) IS (RESPONSE).
    Q2  PURPOSE OF PROGRAM ?
    S2  PROGRAM (<ID>) IS TO (RESPONSE).
    Q3  INITIAL DATA?
    S3  INITIAL DATA OF PROGRAM (<ID>) IS (RESPONSE).
    Q4  PROCESSING LOGIC?
    S4  PROCESSING LOGIC OF PROGRAM (<ID>) IS TO (RESPONSE).
    Q5  FINAL DATA?
    S5  FINAL DATA OF PROGRAM (<ID>) IS (RESPONSE).
    Q6  REFERENCES?
    S6  REFERENCES FOR PROGRAM (<ID>) ARE (RESPONSE).
<SEG DECL>
    NO QUESTION
    NO STATEMENT
<SYN DC2> (FOR EACH IDENTIFIER DECLARED)
    Q1  SYNONYM (<ID>) TO (<ID>) AT (COORDINATE)?
    S1  SYNONYM (<ID>) TO (<ID>) AT (COORDINATE) IS (RESPONSE).
<T CELL ASS>
    Q1  VALUE OF (<ID>) AT (COORDINATE)?
    S1  VALUE OF (<ID>) AT (COORDINATE) IS (RESPONSE).
<T DECL?>
    Q1  (<ID>) AT (COORDINATE)?
    S1  (<ID>) AT (COORDINATE) IS (RESPONSE).
<WHILE ST>
    Q1  PURPOSE OF WHILE STATEMENT (COORDINATES)?
    S1  WHILE STATEMENT (COORDINATES) IS TO (RESPONSE).
    Q2  WHILE CONDITION AT (COORDINATE) ?
    S2  WHILE CONDITION AT (COORDINATE) TESTS (RESPONSE).
```

Figure 2.–Skeleton question/statements for documentation units.

For convenience, we introduce a new basic symbol PROGRAM and the redefinition

⟨PROGRAM⟩ ::=
PROGRAM ⟨ID⟩ ⟨STATEMENT⟩

which permits the naming of programs and reference to documentation units by line numbers, qualified by program names.

CANONICAL DATA FILE

For convenience in documentation processing, we define a canonical data file as consisting of a record for each documentation unit of a program or procedure declaration. Its function is not only to store relationships between various syntactic entities but also to provide data for driving interrogation, report generation, and query processing concerning the program or procedure. Each record describes three properties of the documentation unit: its coordinates in the program text, its syntactic type, and an identifier list. The coordinates are the first and the last lines of the documentation unit (which may be the same when text is contained in a single line). The syntactic type is the entity identified as a documentation unit in figure 1. The identifier list depends on the syntactic type–denoting identifiers which are declared, assigned values, used in assigning values, used in control logic, etc.

It is clear that a deeper syntactical structure, described only informally here, is relevant below the generic level of documentation unit. For example, the identifier list itself is definable in terms of productions within a documentation unit, and such productions determine whether each identifier is being declared, assigned a value, used in a computation, used in control logic, etc. Thus the additional interpretation rules required for documentation processing are distributed throughout the syntax, all the way down to the identifier level, but are not discussed in detail now.

SYNTAX-DIRECTED INTERROGATION AND RESPONSE EDITING

We consider an automatic interrogation process, which uses the canonical data file to complete prestored skeleton questions with program text coordinates and/or identifiers. The interrogation process proceeds through the file, a record at a time, and generates a series of questions from each record, depending on the syntactic type and identifier list found therein. The responses to such questions, made by the programmer, are indexed to the records which generated them.

A set of skeleton questions associated with different documentation units in PL360 is displayed in figure 2. At the end of each interrogation, the programmer is given a final opportunity to volunteer any additional information.

Associated with each skeleton question in figure 2 is a skeleton statement which contains the programmer's response to that question as one of its parts. These statements, filled in with responses and other data from the canonical data file, as shown, represent basic unit messages which can be assembled into reports and query replies.

The construction of skeleton questions and skeleton statements to elicit and edit programmer responses is a substantial and still open problem. It is evident that careless questioning can bury programmers in questionnaires and alienate them to the whole idea. Limited experience (refs. 3 and 4) has indicated that skeleton questions should be terse and highly selective. An involved question, which seems reasonable to read once or twice, can have a very negative effect on a responder when repeated many times, even though this kind of question requires no more effort to answer than a terse one. Thus a first principle in question construction is that the burden of understanding what the question means must be put into a separate orientation course, outside the interrogation itself, and the questionnaires must be kept as short as practicable.

A second principle in question formation is that program text itself must be depended upon for later programmer reference. The questions and responses are intended to illuminate the program text, not to replace it. Otherwise, questions become too involved with points in plain sight in the program text.

Similarly, the order of questioning is also important. Some experience indicates that a "top-down" sequence is a better basis for questioning than "bottom-up." Fortunately, due to the structure of PL360, interrogating documentation units in the order in which their starting text appears gives a top-down approach, which seems easy to follow and reference from both syntactic and typographical viewpoints.

It has been suggested that the matter of question formation might be related to the problem of proving the correctness of programs. Naur (ref. 5) discusses an approach to proving the correctness of programs by "general snapshots," e.g., the state of all variables at various points in programs. These general snapshots could be defined at the entries to and exits from documentation units. This raises the possibility of forming such questions as: "What variables can be modified in this documentation unit?" and "What relationships between the variables must hold (a) on entry to or (b) on exit from this documentation unit?"

At the moment, no suitable way of forming such deeper questions for automatic interrogation is known. But this is an area where future progress may be possible.

DOCUMENTATION PRODUCTS

As already noted, two principal documentation products are—

Documentation reports: complete descriptions, in a prescribed format, of programs or procedures.

Query replies: partial reports in response to queries made by programmers familiar with programs or procedures to probe specific details.

It is to be noted that both interrogation and query reply processing lend themselves to conversational techniques (ref. 4). The canonical data file can be used to drive a conversational interrogation of a programmer quite directly. Similarly, the same file, with an associated file of indexed programmer responses, can be used to generate "computer-assisted instruction courses" automatically when the subjects are particular PL360 programs or procedures.

It should be emphasized that the documentation discussed is addressed to a programmer who understands PL360 and will be reading the PL360 text concurrently. The documentation products are not intended to replace this text as the ultimate authority of what the program does. Rather these products are intended to supplement the program text with perspective, motivation, identifier meanings, processing rationale, etc. In this way it is expected to increase the power and precision with which a programmer can deal with the program text, to modify it, to verify its functional logic, and to assure the integrity of a programming system containing it.

The documentation products will not themselves fill needs of higher level documentation related to user directions, instruction manuals, etc. However, technical writers concerned with such higher level documentation should find these products extremely useful as source material.

DOCUMENTATION REPORTS

We define a standard documentation report with three parts:

(1) Program text
(2) Edited responses
(3) Cross-references

The program text is the relisted, labeled text used in interrogation. The typographical arrangement of this relisting itself shows the overall syntactic structure of the program and/or procedures.

The edited responses, listed in the same order as the questions which generated them, proceed through the text in a systematic way so that one can refer back and forth between the relisted text and the responses efficiently in reading them together. It is expected that the program text and edited responses will be read together by programmers. It would be feasible to intersperse the responses, as comments, in the text, but it seems more desirable to treat them as separate documents with easy interference facilities.

In fact, as a programmer becomes more familiar with the details of a program, the presence of extensive comments tends to inhibit the visual perception of program structure and logic: first, by simply taking up space and expanding the size of material to be looked at; and second, by interrupting and masking typographical features corresponding to the syntactical structure of the program.

The cross-references assemble identifer, function, and procedure usage into cross-reference tables. Identifier usage in the text is categorized into "declared," "assigned," "used in assignments," and "used in control." It is expected that these cross-references serve most of a programmer's needs for evaluating and/or modifying small programs or procedures; for example, to assure that all implications of a changed data declaration are accounted for.

Note that such cross-references can be assembled directly by interpretation rules during program analysis at the time various productions are recognized but then are referred to only informally here.

One particular use of cross-references in PL360 of some potential importance is the recognition of commonality of data references. In particular, the use of identifiers synonymous with hardware registers, which add considerably to the readability of PL360 text, can be found with the aid of such cross-references.

QUERY REPLIES

It is possible to generate a documentation report for any size system of programs or procedures, of course, as a sequence of documentation reports of all its component procedures and programs. However, where documentation reports for a small procedure can be examined rather easily for any information in it, the human eye and mind cannot take in the scope and details of a large system so readily. Thus simply listing a documentation report of a large system, while perhaps of value as a hard-copy reference, is still unsatisfactory for a programmer seeking to understand, modify, or augment a procedure interacting with many other parts of the system. This may be even more critical for a system manager, who is trying to verify the correctness of a new procedure and to assure that no ill effects occur in the system in accepting that new procedure.

This very problem has motivated the foregoing acquisition of documentation as responses to specific questions so that the documentation can be indexed down to the statement and identifer level. Thus the documentation in a large system can be enhanced by the capability for automatic selective retrieval and analysis of documentation. In this sense, the problem of a programmer is not so different from other information systems where data must be stored for retrieval from many points of interest.

A query language for accessing the type of data in these documentation files can be readily imagined and is not defined in detail here. Its output could simply be a selection of edited responses, as defined above. As already noted, such a query capability would lend itself well to conversational methods of programmer access to the documentation. Its capabilities should include, for any given documentation unit, finding identifier usages, extracting "purpose of" responses for all its members, identifying all branch points, and locating all references to keywords in responses.

PROGRAMMER ADAPTATION

In the final analysis, it is expected that the important issues in making such a syntax-directed documentation process effective will be the soundness of the structural approach,

```
PROCEDURE MAGICSQUARE (R6);
COMMENT  THIS PROCEDURE ESTABLISHES A MAGIC SQUARE OF ORDER N, IF N IS
   ODD AND 1 < N < 16. X IS THE MATRIX IN LINEARIZED FORM. REGISTERS
   RO...R6 ARE USED, AND REGISTER RO INITIALLY CONTAINS THE PARAMETER
   N. ALGORITHM 118 COMM. ACM 5 (AUG. 1962) ;
BEGIN SHORT INTEGER NSQR;
   INTEGER REGISTER N SYN RO, I SYN R1, J SYN R2, XX SYN R3,
   IJ SYN R4, K SYN R5;
   NSQR := N; R1 := N * NSQR; NSQR := R1;
   I := N + 1 SHRL 1; J := N;
   FOR K := 1 STEP 1 UNTIL NSQR DO
   BEGIN XX := I SHLL 6; IJ := J SHLL 2 + XX; XX := X(IJ);
      IF XX ¬= 0 THEN
      BEGIN I := I - 1; J := J - 2;
         IF I < 1 THEN I := I + N;
         IF J < 1 THEN J := J + N;
         XX := I SHLL 6; IJ := J SHLL 2 + XX;
      END;
      X(IJ) := K;
      I := I + 1; IF I > N THEN I := I - N;
      J := J + 1; IF J > N THEN J := J - N;
   END;
END
```

Figure 3.—Procedure Magicsquare (ref. 1, p. 53).

```
1   PROCEDURE MAGICSQUARE (R6);
2     BEGIN
3       SHORT INTEGER NSQR;
4       INTEGER REGISTER N SYN RO, I SYN R1, J SYN R2, XX SYN R3,
        IJ SYN R4, K SYN R5;
5       NSQR := N;
6       R1 := N * NSQR;
7       NSQR := R1;
8       I := N + 1 SHRL 1;
9       J := N;
10      FOR K := 1 STEP 1 UNTIL NSQR DO
11        BEGIN
12          XX := I SHLL 6;
13          IJ := J SHLL 2 + XX;
14          XX := X(IJ);
15          IF XX ¬= 0 THEN
16            BEGIN
17              I := I - 1;
18              J := J - 2;
19              IF I < 1 THEN
20                I := I + N;
21              IF J < 1 THEN
22                J := J + N;
23              XX := I SHLL 6;
24              IJ := J SHLL 2 + XX;
25            END;
26          X(IJ) := K;
27          I := I + 1;
28          IF I > N THEN
29            I := I - N;
30          J := J + 1;
31          IF J > N THEN
32            J := J - N;
33        END;
34    END
```

Figure 4.—Syntax-defined listing of Magicsquare.

rather than niceties of question phrasing or report formation. This is because programmers, as human beings, have a large capacity to adapt to matters of English usage but a small capacity to deal with extended program syntax structures in detail.

In the interrogation process, programmers will soon learn how to phrase their responses gracefully in matters of English usage such as parts of speech and tense simply by examining the edited responses which their answers generate. Also, they will learn how the details of their rationale should be allocated among responses by experience in interrogation and by examining the resulting documentation reports. It will still take ability to document programs, but an ability which is adapted to the automatic process being used to acquire and dispense the documentation.

For example, a programmer new to the process may respond to a question about a block by going into the details of statements inside the block. After going through several interrogations and realizing he will be questioned about the included statements later anyway, he will learn to confine his response about the block to the block as a unit. Similarly, by learning that conditions for branching IF statement will be taken up separately, a programmer, following the treatment of the IF statement as a unit, will address his response to the IF statement itself.

In using the documentation of others, a programmer, from his own experience as an originating programmer, will be aware of the questions which generated the responses. He will know, simply by examining program text himself, what questions were asked about any documentation unit or identifier he may be interested in and where they were asked. Thus he can exert considerable intelligence in selective queries of documentation files.

AN EXAMPLE

Figures 3 to 9 simulate the foregoing methods on a sample PL360 procedure, found in reference 1, showing the relisting and interrogation, the canonical data file, a set of responses, a documentation report, and, finally, a set of query replies.

```
COOR-     DOC.    IDENTIFIERS
DINATES   UNIT
1,34      12      MAGICSQUARE, R6
2,34       1
3,3       18      NSQR
4,4       16      N, I, J, XX, IJ, K
5,5       17      NSQR, N
6,6       10      R1, N, NSQR
7,7       17      NSQR, R1
8,8       10      I, N
9,9       10      J, N
10,33      3      K, NSQR
11,33      1
12,12     10      XX, I
13,13     10      IJ, J, XX
14,14     10      XX, X(IJ)
15,25      9      XX
16,25      1
17,17     10      I, I
18,18     10      J, J
19,20      9      I
20,20     10      I, I, N
21,22      9      J
22,22     10      J, J, N
23,23     10      XX, I
24,24     10      IJ, J, XX
26,26     17      X, IJ, K
27,27     10      I, I
28,29      9      I, N
29,29     10      I, I, N
30,30     10      J, J
31,32      9      J, N
32,32     10      J, J, N
```

Figure 5.–Canonical data of Magicsquare.

```
FILE KEY  QUESTION

1,34,1    AUTHOR OF PROCEDURE MAGICSQUARE?
1,34,2    PURPOSE OF PROCEDURE MAGICSQUARE?
1,34,3    INITIAL DATA?
1,34,4    PROCESSING LOGIC?
1,34,5    FINAL DATA?
1,34,6    REFERENCES?
2,34,1    PURPOSE OF BLOCK 2,34 ?
3,3,1     NSQR AT 3 ?
4,4,1     N AT 4 ?
4,4,2     I AT 4 ?
4,4,3     J AT 4 ?
4,4,4     XX AT 4 ?
4,4,5     IJ AT 4 ?
4,4,6     K AT 4 ?
5,5,1     VALUE OF NSQR AT 5 ?
6,6,1     VALUE OF R1 AT 6 ?
7,7,1     VALUE OF NSQR AT 7 ?
8,8,1     VALUE OF I AT 8 ?
9,9,1     VALUE OF J AT 9 ?
10,33,1   PURPOSE OF FOR STATEMENT 10,33 ?
10,33,2   FOR CONDITION AT 10 ?
11,33,1   PURPOSE OF BLOCK 11,33 ?
12,12,1   VALUE OF XX AT 12 ?
13,13,1   VALUE OF IJ AT 13 ?
14,14,1   VALUE OF XX AT 14 ?
15,25,1   PURPOSE OF IF THEN STATEMENT 15,25 ?
15,25,2   IF CONDITION AT 15 ?
16,25,1   PURPOSE OF BLOCK 16,25 ?
17,17,1   VALUE OF I AT 17 ?
18,18,1   VALUE OF J AT 18 ?
19,20,1   PURPOSE OF IF THEN STATEMENT 19,20 ?
19,20,2   IF CONDITION AT 19 ?
20,20,1   VALUE OF I AT 20 ?
21,22,1   PURPOSE OF IF THEN STATEMENT 21,22 ?
21,22,2   IF CONDITION AT 21 ?
22,22,1   VALUE OF J AT 22 ?
23,23,1   VALUE OF XX AT 23 ?
24,24,1   VALUE OF IJ AT 24 ?
26,26,1   VALUE OF X(IJ) AT 26 ?
27,27,1   VALUE OF I AT 27 ?
28,29,1   PURPOSE OF IF THEN STATEMENT 28,29 ?
28,29,2   IF CONDITION AT 28 ?
29,29,1   VALUE OF I AT 29 ?
30,30,1   VALUE OF J AT 30 ?
31,32,1   PURPOSE OF IF THEN STATEMENT 31,32 ?
31,32,2   IF CONDITION AT 31 ?
32,32,1   VALUE OF J AT 32 ?
1,34,7    ANY FURTHER COMMENTS ?
```

Figure 6.–Syntax-defined interrogation for Magicsquare.

Figure 3 is a PL360 procedure named by Magicsquare, just as formulated by Wirth (ref. 1), including the typography. This procedure, adapted from an ALGOL procedure published in the Algorithm department of *Communications* of the ACM (ref. 6), builds magic squares of odd order n when $1 < n < 16$.

Figure 4 is a syntax-defined and labeled relisting of the same PL360 procedure Magicsquare, less comments, with its typography determined by the rules already given for recognizing lines and their indentation. This relisting is independent of the typography of the program text in figure 3. It is expected that such a standard yet flexible form of program text will, in itself, help programmers read each other's programs.

Figure 5 shows the contents of the canonical data file generated by procedure Magicsquare. All further interrogation, response editing, and other documentation processing will use this canonical data file and not the program text. This particular file contains 31 records with some 157 separate items of data in them: two coordinates, a syntactic type, and an average of about two identifiers per record.

Figure 6 gives the syntax-directed interrogation of Magicsquare, using the canonical data file and the skeleton questions of figure 2. There are 48 questions in all, which refer to the coordinates of the relisted program text and represent a systematic coverage of the text. A final question gives a programmer an opportunity to volunteer additional information not already solicited by the previous questions.

Figure 7 contains a set of responses to the interrogation of figure 6. There is a file key associated with each question, which is used to label responses so that they may be indexed to the proper questions. The author has presumed to speak for "programmer Wirth" in constructing these responses.

Figure 8 provides a resulting documentation report in the three sections described already: source code, edited responses, and cross-references. For a short procedure or program such as this one, it is expected that a documentation report itself will be sufficient to allow a programmer to find out anything he wants to know about the procedure or program.

```
FILE KEY  RESPONSE

1,34,1    NIKLAUS WIRTH, STANFORD UNIVERSITY, DECEMBER 20, 1966.
1,34,2    ESTABLISH A MAGIC SQUARE OF ORDER N, IF N IS ODD AND 1 < N < 16.
1,34,3    THE ORDER, N, OF THE MAGIC SQUARE DESIRED.
1,34,4    FILL SQUARE MATRIX WITH SUCCESSIVE INTEGERS ALONG CERTAIN DIAGONALS
          AND THEIR EXTENSIONS TO ENSURE MAGIC SQUARE PROPERTY. THE MATRIX TO
          BE FILLED IS ASSUMED TO CONTAIN ALL ZEROES INITIALLY.
1,34,5    THE MAGIC SQUARE X AS A MATRIX IN LINEARIZED FORM.
1,34,6    ALGORITHM 118, COMM ACM, AUGUST 1962, P 436; M. KRAITCHIK.
          MATHEMATICAL RECREATIONS, P 149.
2,34,1    CARRY OUT THE PROCEDURE MAGICSQUARE.
3,3,1     THE NUMBER OF ENTRIES IN THE MAGIC SQUARE.
4,4,1     THE ORDER (NUMBER OF ROWS AND COLUMNS) OF THE MAGIC SQUARE.
4,4,2     THE ROW INDEX FOR THE NEXT INTEGER VALUE GOING INTO THE MAGIC SQUARE.
4,4,3     THE COLUMN INDEX FOR THE NEXT INTEGER VALUE GOING INTO THE MAGIC
          SQUARE.
4,4,4     INTERMEDIATE VALUE IN X OFFSET CALCULATION AND TO TEST X VALUE FOR
          ZERO.
4,4,5     THE X OFFSET FOR ROW I, COLUMN J OF MAGIC SQUARE.
4,4,6     THE NEXT INTEGER VALUE GOING INTO MAGIC SQUARE.
5,5,1     INTERMEDIATE VALUE N FOR NSQR.
6,6,1     TEMPORARY STORAGE OF NSQR
7,7,1     FINAL VALUE OF NSQR, THE NUMBER OF ENTRIES IN THE MAGIC SQUARE.
8,8,1     INITIAL VALUE FOR I.
9,9,1     INITIAL VALUE FOR J.
10,33,1   FILL MAGIC SQUARE WITH INTEGERS.
10,33,2   STEP K THROUGH INTEGERS FROM 1 TO NSQR, WHICH WILL APPEAR IN THE
          MAGIC SQUARE.
11,33,1   FIND CORRECT LOCATION IN MAGIC SQUARE FOR INTEGER K.
12,12,1   X OFFSET FOR ROW I OF MAGIC SQUARE.
13,13,1   X OFFSET FOR ROW I AND COLUMN J OF MAGIC SQUARE.
14,14,1   CURRENT VALUE OF POINT I, J IN MAGIC SQUARE.
15,25,1   BEGIN NEW DIAGONAL IF CURRENT DIAGONAL IS ALREADY FILLED.
15,25,2   IS DIAGONAL FILLED (AN INTEGER ALREADY STORED AT POINT I,J)?
16,25,1   FIND STARTING LOCATION FOR NEXT DIAGONAL TO BE FILLED.
17,17,1   NEW ROW INDEX OF STARTING LOCATION.
18,18,1   NEW COLUMN INDEX OF STARTING LOCATION.
19,20,1   RESTORE ROW INDEX TO CORRECT RANGE, IF NECESSARY.
19,20,2   IS ROW INDEX OUT OF RANGE?
20,20,1   ROW INDEX IN CORRECT RANGE.
21,22,1   RESTORE COLUMN INDEX TO CORRECT RANGE, IF NECESSARY.
21,22,2   IS COLUMN INDEX IN CORRECT RANGE ?
22,22,1   COLUMN INDEX IN CORRECT RANGE.
23,23,1   X OFFSET FOR ROW I OF MAGIC SQUARE.
24,24,1   X OFFSET FOR ROW I AND COLUMN J OF MAGIC SQUARE.
26,26,1   FINAL INTEGER VALUE AT POINT I, J IN MAGIC SQUARE.
27,27,1   ROW INDEX STEPPED ALONG DIAGONAL.
28,29,1   RESTORE ROW INDEX TO CORRECT RANGE, IF NECESSARY.
28,29,2   IS ROW INDEX IN CORRECT RANGE?
29,29,1   ROW INDEX IN CORRECT RANGE.
30,30,1   COLUMN INDEX STEPPED ALONG DIAGONAL.
31,32,1   RESTORE COLUMN INDEX TO CORRECT RANGE, IF NECESSARY.
31,32,2   IS COLUMN INDEX IN CORRECT RANGE?
32,32,1   COLUMN INDEX IN CORRECT RANGE.
1,34,7    NO.
```

Figure 7.–Interrogation responses for Magicsquare.

```
 MAGICSQUARE PROGRAM TEXT

 1  PROCEDURE MAGICSQUARE (R6);
 2    BEGIN
 3      SHORT INTEGER NSQR;
 4      INTEGER REGISTER N SYN R0, I SYN R1, J SYN R2, XX SYN R3,
        IJ SYN R4, K SYN R5;
 5      NSQR := N;
 6      R1 := N * NSQR;
 7      NSQR := R1;
 8      I := N + 1 SHRL 1;
 9      J := N;
10      FOR K := 1 STEP 1 UNTIL NSQR DO
11        BEGIN
12          XX := I SHLL 6;
13          IJ := J SHLL 2 + XX;
14          XX := X(IJ);
15          IF XX ¬= 0 THEN
16            BEGIN
17              I := I - 1;
18              J := J - 2;
19              IF I < 1 THEN
20                I := I + N;
21              IF J < 1 THEN
22                J := J + N;
23              XX := I SHLL 6;
24              IJ := J SHLL 2 + XX;
25            END;
26          X(IJ) := K;
27          I := I + 1;
28          IF I > N THEN
29            I := I - N;
30          J := J + 1;
31          IF J > N THEN
32            J := J - N;
33        END;
34    END

 MAGICSQUARE EDITED RESPONSES

FILE KEY  EDITED RESPONCE

1,34,1    AUTHOR OF PROCEDURE MAGICSQUARE IS NIKLAUS WIRTH, STANFORD UNIVERSITY.
          DECEMBER 20, 1966.
1,34,2    PROCEDURE MAGICSQUARE IS TO ESTABLISH A MAGIC SQUARE OF ORDER N, IF N
          IS ODD AND 1 < N < 26.
1,34,3    INITIAL DATA OF PROCEDURE MAGICSQUARE IS THE ORDER, N, OF THE MAGIC
          SQUARE DESIRED.
1,34,4    PROCESSING LOGIC OF PROCEDURE MAGICSQUARE IS TO FILL SQUARE  MATRIX
          WITH SUCESSIVE INTEGERS ALONG CERTAIN DIAGONALS AND THEIR EXTENSIONS
          TO ENSURE MAGIC SQUARE PROPERTY. THE MATRIX TO BE FILLED IS ASSUMED
          TO CONTAIN ALL ZEROES INITIALLY.
1,34,5    FINAL DATA OF PROCEDURE MAGICSQUARE IS THE MAGIC SQUARE X AS A MATRIX
          IN LINEARIZED FROM.
1,34,6    REFERENCES FOR MAGICSQUARE ARE ALGORITHM 118, COMM ACM, AUGUST 1962,
          P 436; M. KRAITCHIK, MATHEMATICAL RECREATIONS, P 149.
2,34,1    BLOCK 2,34 IS TO CARRY OUT THE PROCEDURE MAGICSQUARE.
3,3,1     NSQR AT 3 IS THE NUMBER OF ENTRIES IN THE MAGIC SQUARE.
4,4,1     N AT 4 IS THE ORDER (NUMBER OF ROWS OR COLUMNS) OF THE MAGIC SQUARE.
4,4,2     I AT 4 IS THE ROW INDEX FOR THE NEXT INTEGER VALUE GOING INTO THE
          MAGIC SQUARE.
4,4,3     J AT 4 IS THE COLUMN INDEX FOR THE NEXT INTEGER VALUE GOING INTO THE
          MAGIC SQUARE.
4,4,4     XX AT 4 IS INTERMEDIATE VALUE IN X OFFSET CALCULATION AND TO TEST X
          VALUE FOR ZERO.
4,4,5     IJ AT 4 IS THE X OFFSET FOR ROW I, COLUMN J OF MAGIC SQUARE.
4,4,6     K AT 4 IS THE NEXT INTEGER VALUE GOING INTO THE MAGIC SQUARE.
5,5,1     VALUE OF NSQR AT 5 IS INTERMEDIATE VALUE FOR NSQR.
6,6,1     VALUE OF R1 AT 6 IS TEMPORARY STORAGE OF NSQR.
7,7,1     VALUE OF NSQR AT 7 IS FINAL VALUE OF NSQR, THE NUMBER OF ENTRIES IN
          THE MAGIC SQUARE.
8,8,1     VALUE OF I AT 8 IS INITIAL VALUE FOR I.
9,9,1     VALUE OF J AT 9 IS INITIAL VALUE FOR J.
10,33,1   FOR STATEMENT 10,33 IS TO FILL MAGIC SQUARE WITH INTEGERS.
10,33,2   FOR CONDITION AT 10 IS TO STEP K THROUGH INTEGERS FROM 1 TO NSQR.
          WHICH WILL APPEAR IN THE MAGIC SQUARE.
11,33,1   BLOCK 11,33 IS TO FIND CORRECT LOCATION IN MAGIC SQUARE FOR INTEGER K.
12,12,1   VALUE OF XX AT 12 IS X OFFSET FOR ROW I OF MAGIC SQUARE.
13,13,1   VALUE OF IJ AT 13 IS X OFFSET FOR ROW I AND COLUMN J OF MAGIC SQUARE.
14,14,1   VALUE OF XX AT 14 IS CURRENT VALUE OF POINT I, J IN MAGIC SQUARE.
15,25,1   IF THEN STATEMENT 15,25 IS TO BEGIN NEW DIAGONAL IF CURRENT DIAGONAL
          IS ALREADY FILLED.
15,25,2   IF CONDITION AT 15 TESTS IS DIAGONAL FILLED (AN INTEGER ALREADY STORED
          AT POINT I,J)?
16,25,1   BLOCK 16,25 IS TO FIND STARTING LOCATION FOR NEXT DIAGONAL TO BE
          FILLED.
17,17,1   VALUE OF I AT 17 IS NEW ROW INDEX OF STARTING LOCATION.
18,18,1   VALUE OF J AT 18 IS NEW COLUMN INDEX OF STARTING LOCATION.
19,20,1   IF THEN STATEMENT 19,20 IS TO RESTORE ROW INDEX TO CORRECT RANGE, IF
          NECESSARY.
19,20,2   IF CONDITION AT 19 TESTS IS ROW INDEX OUT OF RANGE?
20,20,1   VALUE OF I AT 20 IS ROW INDEX IN CORRECT RANGE.
21,22,1   IF THEN STATEMENT 21,22 IS TO RESTORE COLUMN INDEX TO CORRECT RANGE.
          IF NECESSARY.
21,22,2   IF CONDITION AT 21 TESTS IS COLUMN INDEX IN CORRECT RANGE?
22,22,1   VALUE OF J AT 22 IS COLUMN INDEX IN CORRECT RANGE.
23,23,1   VALUE OF XX AT 23 IS X OFFSET FOR ROW I OF MAGIC SQUARE.
24,24,1   VALUE OF IJ AT 24 IS X OFFSET FOR ROW I OF COLUMN J OF MAGIC SQUARE.
26,26,1   VALUE OF X(IJ) AT 26 IS FINAL INTEGER VALUE AT POINT I, J IN MAGIC
          SQUARE.
27,27,1   VALUE OF I AT 27 IS ROW INDEX STEPPED ALONG DIAGONAL.
28,29,1   IF THEN STATEMENT 28,29 IS TO RESTORE ROW INDEX TO CORRECT RANGE. IF
          NECESSARY.
28,29,2   IF CONDITION AT 28 TESTS IS ROW INDEX IN CORRECT RANGE?
29,29,1   VALUE OF I AT 29 IS ROW INDEX IN CORRECT RANGE.
30,30,1   VALUE OF J AT 30 IS COLUMN INDEX STEPPED ALONG DIAGONAL.
31,32,1   IF THEN STATEMENT 31,32 IS TO RESTORE COLUMN INDEX TO CORRECT RANGE,
          IF NECESSARY.
31,32,2   IF CONDITION AT 31 TESTS IS COLUMN INDEX IN CORRECT RANGE?
32,32,1   VALUE OF J AT 32 IS COLUMN INDEX IN CORRECT RANGE.
1,34,7    NO FURTHER COMMENTS.

MAGICSQUARE CROSS REFERENCES

      DATA CROSS REFERENCES

          I: DC 4: AS 6,8,17,20,27,29: UA 7,12,17,20,23,27,29: UC 19,28:
          IJ: DC 4; AS 13,24; UA 14,26;
          J: DC 4; AS 9,18,22,30,31; UA 13,18,22,24,30,31: UC 21,31:
          K: DC 4; AS 10; UA 10,26; UC 10;
          MAGICSQUARE: DC 1:
          N: DC 4; UA 5,6,8,9,20,22,29,32; UC 28,31;
          NSQR: DC 3; AS 5,7; UA 6; UC 10;
          R0: UA 5,6,8,9,20,22,29,32: UC 28,31:
          R1: AS 6,8,17,20,27,29; UA 7,12,17,20,23,27,29; UC 19,28:
          R2: AS 9,18,22,30,31; UA 13,18,22,24,30,31; UC 21,31;
          R3: AS 12,14,23; UA 13,24; UC 15;
          R4: AS 13,24; UA 14,26;
          R5: AS 10; UA 10,26; UC 10:
          R6: UC 1:
          X(IJ): AS 26; UA 14;
          XX: DC 4: AS 12,14,23; UA 13,24,; UC 15;

      FUNCTION CROSS REFERENCES

          NO FUNCTION CROSS REFERENCES.

      PROCEDURE CROSS REFERENCES

          NO PROCEDURE CROSS REFERENCES.
```

Figure 8.–Documentation report for Magicsquare.

```
   QUERY: ALL REFERENCES TO K

     QUERY REPLY:

4,4,6     K AT 4 IS THE NEXT INTEGER VALUE GOING INTO THE MAGIC SQUARE.
10,33,1   FOR STATEMENT 10,33 IS TO FILL MAGIC SQUARE WITH INTEGERS.
10,33,2   FOR CONDITION AT 10 IS TO STEP K THROUGH INTEGERS FROM 1 TO NSQR.
          WHICH WILL APPEAR IN THE MAGIC SQUARE.
26,26,1   VALUE OF X(IJ) AT 26 IS FINAL INTEGER VALUE AT POINT I,J IN MAGIC
          SQUARE.

   QUERY: ALL BRANCHES

     QUERY REPLY:
10,33,2   FOR CONDITION AT 10 IS TO STEP K THROUGH INTEGERS FROM 1 TO NSQR.
          WHICH WILL APPEAR IN THE MAGIC SQUARE.
15,25,2   IF CONDITION AT 15 TESTS IS DIAGONAL FILLED (AN INTEGER ALREADY STORED
          AT POINT I,J) ?
19,20,2   IF CONDITION AT 19 TESTS IS ROW INDEX OUT OF RANGE ?
21,22,2   IF CONDITION AT 21 TESTS IS COLUMN INDEX IN CORRECT RANGE ?
28,29,2   IF CONDITION AT 28 TEST IS ROW INDEX IN CORRECT RANGE ?
31,32,2   IF CONDITION AT 31 TESTS IS COLUMN INDEX IN CORRECT RANGE ?

   QUERY: ALL REFERENCES TO KEYWORD 'DIAGONAL' IN RESPONSES

     QUERY REPLY:
1,34,4    PROCESSING LOGIC OF PROCEDURE MAGICSQUARE IS TO FILL SQUARE MATRIX WITH
          SUCESSIVE INTEGERS ALONG CERTAIN DIAGONALS AND THEIR EXTENSIONS TO
          ENSURE MAGIC SQUARE PROPERTY. THE MATRIX TO BE FILLED IS ASSUMED TO
          CONTAIN ALL ZEROES INITIALLY.
15,25,1   IF THEN STATEMENT 15,25 IS TO BEGIN NEW DIAGONAL IF CURRENT DIAGONAL
          IS ALREADY FILLED.
15,25,2   IF CONDITION AT 15 TESTS IS DIAGONAL FILLED (AN INTEGER ALREADY STORED
          AT POINT I,J)?
16,25,2   BLOCK 16,25 IS TO FIND STARTING LOCATION FOR NEXT DIAGONAL TO BE
          FILLED.
27,27,1   VALUE OF I AT 27 IS ROW INDEX STEPPED ALONG DIAGONAL.
30,30,1   VALUE OF J AT 30 IS COLUMN INDEX STEPPED ALONG DIAGONAL.

   QUERY: ALL USES IN ASSIGNMENTS OF IJ

     QUERY REPLY:

14,14,1   VALUE OF XX AT 14 IS CURRENT VALUE OF POINT I,J IN MAGIC SQUARE.
26,26,1   VALUE OF X(IJ) AT 26 IS FINAL INTEGER VALUE AT POINT I,J IN MAGIC
          SQUARE.
```

Figure 9.–Some query replies for Magicsquare.

Figure 9 indicates how certain queries might be used to probe more specifically into the procedure via syntactic, identifier, or response keyword criteria. Note in each case a subset of the edited responses of a full documentation report is simply compiled according to a query condition.

In all these listings, the file keys have been listed to make the storage/retrieval process transparent. In practice, they could be suppressed in documentation reports and query replies.

ACKNOWLEDGMENTS

The author acknowledges useful suggestions from referees, particularly on some specifics of PL360 and on the automatic formation of questions. The relationship between proving the correctness of programs and the interrogation process was suggested by a referee.

REFERENCES

1. Wirth, N.: PL360, A Programming Language for the 360 Computers. *J. Ass. Computing Machinery* **15**: 37-74, Jan. 1968.
2. Wirth, N.; and Weber, H. Euler: A Generalization of ALGOL, and Its Formal Definition: Pt. I. *Commun. Ass. Computing Machinery* 9(1): 13-23, Jan. 1966.
3. Mills, H. D.; and Dyer, M.: Evolutionary Systems for Data Processing. *IBM Real-Time Systems Seminar,* Nov. 1966, pp. 1-9.
4. Meadow, C. T.; and Waugh, D. V.: Computer Assisted Interrogation. *Proc. AFIPS 1966 Fall Joint Comput. Conf.* Vol. 29, Spartan Books, Inc., pp. 381-394.

5. Naur, P.: Proof of Algorithms by General Snapshots. *BIT* **6**: 310-316, 1966.
6. Collison, D. M.: Algorithm 118, Magic Square (Odd Order). *Commun. Ass. Computing Machinery* 5(8): 456, Aug. 1962.

DISCUSSION

MEMBER OF THE AUDIENCE: Could you in the running program have asked some questions beforehand, such as what are the ranges of your variables, so that this could be incorporated into the program for error analysis? Could you also use this question-and-answer sort of thing for compiling optimization, so that you actually had a sort of interactive compiler? Do you think these kinds of things might be feasible?

DR. MILLS: Well, I think they probably can. I have not thought about them, but I think that what you say sounds reasonable. I really laid out a very austere kind of thing. It is easy for the mind to boggle at the idea of trying to do computer-assisted interrogation of almost any subject. The computer programs are particularly well structured. I mean we can actually define the syntax. But doing this in other areas may be far-fetched.

MEMBER OF THE AUDIENCE: How long would it take to develop this system?

DR. MILLS: Well, what I described here to you is a paper system because we do not have PL360. But I hope I can get a couple of graduate students to do this quickly.

N73-19214

DOCUMENTATION: MOTIVATION AND TRAINING OR AUTOMATION

Melba L. Mouton
NASA Goddard Space Flight Center

One of the eternally discussed problems in almost any technical or administrative setting is the problem of communications. The subject of this symposium indicates the concern about that problem in programming activities. In fact, it indicates not only concern but that many people are involved in trying to do something about it. Because of this deep concern and because of a tendency of many people in and out of electronic data processing (EDP) management to feel that anything and everything can be fully automated, it would be a good idea to consider what can be done to relieve the roadblocks or mental blocks in those areas where automation is not taking care of the basic documentation problem.

In the development of any sizable computational project, it is common to involve all the organizational elements that are related to the project design and use. These organizational, as well as functional, entities usually consist of people called managers, analysts, data analysts, and programmers. For the moment, consider these as the implementers of the project.

Table 1 indicates that for the first version of such a project, considerable attention is given to planning who will be responsible for developing the various manuals that make up

Table 1.—Original Project Documentation

Type of manual	Implementer	Users	Intended audience
Requirements	Analysts Programmers Managers Data analysts	Analysts Programmers Managers Data analysts	Managers
Analysis	Analysts	Analysts Programmers Managers	Programmers
Program specifications	Analysts Programmers	Analysts Programmers	Programmers
User's	Programmers	Data analysts Programmers Analysts	Data analysts
Programmer's	Programmers	Programmers	Programmers

the system documentation. On the left, the various types of manuals, requirements, analysis, program specifications, users, and programmers, are listed. In the next columns, the implementers, the users, and the persons for whom the document was written are listed.

The implementers for the requirements manual should involve all four disciplines; the analysts, programmers, managers, and data analysts are therefore listed. The users are normally a similar group of people; however, it is good practice to consider that it is being written for the manager.

The analysis manual is written by the analyst for the use of programmers, analysts, and managers. It is necessary for, and thus documented for, the programmers. Because of this need, some persons consider it as the first part of the next document, the program specifications manual. In any case, this specifications manual, too, is documented for the programmer and used by both programmers and analysts, the two groups that usually work together in developing it.

Next is the manual that is most often developed and has the least problem in implementer motivation and training, the user's manual. This is usually prepared by programmers for data analysts, but it is quite useful for the analysts and programmers as well.

Last, but not least, is the programmer's manual, written by programmers for programmers and, consequently, written with their knowledge or understanding in mind. This is the only manual for which the author, or the implementer, is the same as the principal audience and user.

Since the persons who need this manual the most are also the persons who are under the greatest pressure to get the program code written and debugged, it is not uncommon that this is the most neglected aspect of system documentation. Until managers, analysts, and programmers begin to make programming equivalent to planning, documenting, coding, documenting, debugging, and documenting, the checked-out, or almost checked-out, code will continue to be considered the end product of a programming effort. However, the goal of programmers' efforts must be both the checked-out code, the communication necessary for the machine, and the checked-out programmer's manual, the communication necessary for people, especially programmers, to continue to develop and maintain the system.

Documentation as a part of planning a program may have a relatively higher initial cost, but if the planning and associated documentation is done well, problems that could cost 10 times as much to solve later may never arise. In addition, for problems that do arise, solutions can be found and implemented more easily. It is difficult to determine who is most responsible for not allowing the time to do adequate documentation of the program, but it is clear that if programmers consistently give time and manpower estimates that do not include documentation, they will not be able to create a document that can give the recognition and reward that should accompany any good programming effort. However, the fault lies equally with those analysts or managers who accept a low bid in time and/or money for an undocumented or a half-documented system rather than a reasonable bid in time and money that allows for an up-to-date, good, clear description of the program. The time not allowed now will be allowed later, not for documentation, but for difficulty in program change and difficulty in the analyst user finding out what is in the system, as the continually changing but undocumented program persists.

For either or both of these reasons, the phase in which everyone is involved in this vital new project changes to what is commonly referred to as the programming maintenance

Table 2.—Documentation for Project Maintenance

Type of manual	Implementers	Users	Intended audience
Requirements			
Analysis			
Program specifications			
User's	Programmers	Data analysts Programmers Analysts	Data analysts
Programmer's			

Table 3.—A Comparison of Preliminary and Final Documentation

Steps in documentation	Preliminary documentation	Final documentation	Automation
Description of the problem	X	X	
Method of solution	X	X	
User instructions	X	X	
Flowchart	X	X	?
Subroutines used	X	X	
Program listing		X	X
Test documentation	X	X	

phase. Table 2 indicates what happens to the manuals that initially may have been adequately done. It only takes a "few" minutes to put in a "simple" change and check it out. Thus the cycle begins of quick change of code, but there is no allowance on the part of managers, analysts, or programmers to keep up what might have been reasonably good documentation. Thus begins the refrain, "it is documented, but it is not up to date." Not allowing the time needed to code and keep supporting documents updated comes at the point in the life of a system when programmers are the only ones seemingly responsible for updating all the relevant documentation, not just the programmer's manual. Thus, standards, guidelines, and, possibly, automation that will really have an impact on the entire problem of keeping computer program documents up to date must be considered.

The biggest problem, however, seems to be in the programmer's manual, and table 3 indicates what could and should be done as preliminary documentation. The only thing that cannot be done then is the listing. As soon as programmers are motivated and trained to plan (including the adoption of useful automatic procedures) and document prior to coding, they will see how well it pays off, especially once it is realized that the manual is written by the time the listing is available. With the source code and the basic documentation available, more automatic procedures may be used for ease in updating and verifying a program and its documentation.

Because a good, descriptive planning flowchart is very important to the development of a program, delaying the generation of flowcharts until after the code is completed is detrimental to programming efforts. At this point, it should be emphasized that current automatic flowchart techniques should use the descriptions from a well-planned flowchart. The emphasis should be on how to produce planning block diagrams, flowcharts, and other documentation that can be updated automatically. A flowchart automatically determined from the source code without such preplanning is almost useless. This is best stated in the following excerpt from reference 1:

> Flowchart—this must be a structural flowchart of the sequential logic and decision points included in the program. Machine-produced flowcharts of the exact programming techniques cannot be used to satisfy this requirement as they merely amount to a listing of the programs and do not briefly and concisely reflect the inherent logical flow of decisions.

This refers, it seems, to "source-code-only" flowcharts.

Figure 1 is a simplified version of some information given in reference 2. It summarizes the importance and attention that must be given to documenting during the whole lifetime of a program or system of programs for good system development and maintenance. Figure 1 simply indicates that documenting, like management, must be considered an integral part of every phase of system development and use.

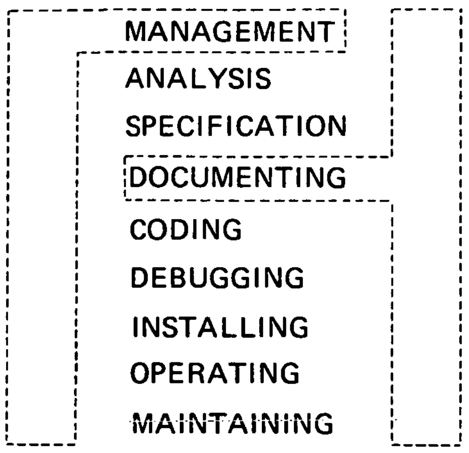


Figure 1.—The relative importance of documenting and management.

REFERENCES

1. Anon.: *Documentation and Program Standards Handbook.* COSMIC, Univ. of Georgia, Nov. 1968.
2. Control Data Corp.: *Seminar in Documentation,* Sept. 1968, p. 13.

DISCUSSION

MEMBER OF THE AUDIENCE: You talked of motivating the programmers. Have you been at all successful in actually motivating them?

MOUTON: I find that it takes more than motivation; it takes a little bit of control on the scheduling. When programmers operate in a very hectic atmosphere, it is very difficult. You can motivate and train them to do a particular job and get them really interested in doing it, but when another job comes across the desk that should have been done yesterday, then you are almost always forced to document after the fact. Where I have seen documentation done properly, it was a requirement. The documentation was done before the job could be put in an operational status.

MEMBER OF THE AUDIENCE: How was that requirement imposed?

MOUTON: The jobs came into the programming shop in the form of an analysis document or something like that and what was to be done had to be specified. The programming branch then developed the program that was turned over to an operational group to run. So before the job was turned over to another group to run, you had to have not just the operating instructions but the complete document. This operation was for the Army Map Service, and because it was sent out of the installation for safekeeping, all of this was done systematically. The time for documentation was included in the original estimate.

PANEL DI$CUSSION

MEMBER OF THE AUDIENCE: I am very sympathetic with the need for documentation, and I am searching for ideas that can be used to persuade the users of programs that documentation is needed. We are here because we are interested in and see the need for it. I have worked for several different companies, Bendix, GE, and RCA, where job-shop programming on a commercial basis for a profit is done. The biggest problem is getting the users of the final outcome of the data to let us document programs and include that in the cost. What arguments can we offer to our managers that they can use in management discussions to get this carried through for us?

PANEL MEMBER: It might help if the managers begin to talk in terms of the overall cost of such a project rather than just the initial cost because I am convinced, even though I have not kept any books, that the overall cost of maintaining or developing new systems because of undocumented systems is much higher than planning and documenting your project carefully.

PANEL MEMBER: I think this is the crux of many of our problems. I think that we do have to get management to recognize the problem that we are faced with. One of the topics to be discussed is what items can be automated to cut some of the costs of programming.

MEMBER OF THE AUDIENCE: Do you think we should recommend standardization for types of program development so that they could be implemented with program automatic techniques? This morning we saw some examples of coded comments that feed into automatic flowcharting. That enables you to develop a higher level of program flowchart of the structural variety rather than the detailed variety. Should we go on in this vein of standardization? Should we attempt to define standards?

PANEL MEMBER: I feel that standards are essential to operating a system, particularly considering the size of the systems that I am used to working with. For example, we have standardized input of time recently. If you have worked with a system for which you input time, you know you have to worry about the kind of time to be used because every programmer puts the time in a different format. We standardized the time and wrote one conversion routine. For all practical purposes, now, in our programs time is no longer a problem. Programmers take this standardized input, pass it on to the program that converts it, and have five or six options as far as the output. There is only one form in which time can go into the system now. It means that when someone is writing a program he can omit this test because we have already checked it out for him, and he does not have to document it and does not have to document the test that was run.

There are many other things that we have standardized in our system. I do not know whether there are standards that you can set up or whether it is worth the trouble for smaller systems.

PANEL MEMBER: I think the imposition of standards depends on how wide an area you are trying to cover with standards. Standardizing within your own particular area could be quite useful.

With reference to flowcharting, if you wrote a planning block diagram or flowchart initially and the programmer could then hand that hand-drawn flowchart and specify what he wants to do to specialized keypunch people and then that could be keypunched in a form that would be the beginning of source code for this document, I think you would then have the plan becoming a part of something that the programmer would be interested in updating.

PANEL MEMBER: Let me point to some of the problems involved in practical standards work. When you are doing something that varies little or where a consensus has almost been achieved, the amount of work involved in setting a standard is not very great. That applies, for instance, to standardizing time. It applies with somewhat less force to standardized flowcharting, symbols, and so forth. But when broader kinds of standards are discussed, there is a greater problem. If someone else tackled Dr. Mills' problem for a different language or even for a PL 360 in a different institution or from a different viewpoint, the chances are there would be a different set of questions.

To achieve a consensus, you need a consensus. To achieve a consensus around a conference table takes forever. In the meanwhile, the project moves on. I think that you have to work on the problem of standardization both theoretically and practically. You should try to reach a formal consensus about what the standard ought to be. At the same time, you should package your own system as best you can and see what kind of workable standards you come up with. But to develop a whole standards book that will enable you to document blindly seems to be almost impossible.

PANEL MEMBER: Another problem is the nature of the individual that you are working with in programming. Inevitably, the best programmer is the brilliant but undisciplined one. He can solve singlehandedly a problem that a whole team has not been able to, but he is incapable of keeping a laboratory notebook or of doing detailed documentation as he goes along. You need something, perhaps an automatic system, to document for him.

PANEL MEMBER: I have to agree with you. In our organization really productive people often do their own programming. We think that they come up with new ideas and new approaches that would be valuable to programmers in general, but they consider the program to be only a tool to an end. We have to help them document their programs so they can be distributed. This brings up a point that we can perhaps discuss here. We have had a lot of discussion about the documentation of entire programs. My question is whether we should concentrate on the documentation of an entire program or of an element that is usable in many programs.

PANEL MEMBER: May I answer that slightly indirectly? Programming is only 20 years old as a profession, and we have certainly not experienced all the problems that it has to offer. It seems to me that our standards have to be somewhat ad hoc and built around the machines we use and the circumstances we use them in.

One problem is that we have no way of objectively deciding when a program is needlessly complex. I think that we must work on measuring the complexity of a program, and I would be interested in Goetz's progress in his work on weighting statements and his attempt to estimate complexity that way. It seems to me that our standards problem is very much predicated on our immaturity or is very much hurt by our immaturity as a profession. We are at the beginning of a growth curve, and 20 years is a terribly short time.

PANEL MEMBER: I am not sure but that we may not have reached a maximum in setting standards. The process of achieving a human consensus has not speeded up very much, but technology is moving faster and faster. I am inclined to think that the time to standardize is now. We are enthusiastic, we are rich, there are a lot of things worth standardizing, and we are trying hard to do it. I think it is going to be increasingly difficult to standardize in the future.

MEMBER OF THE AUDIENCE: I am a programmer, and I would like to take the side of the programmer here and say that perhaps we are going in the wrong direction. Programming is becoming ever more complex. For example, it takes a newsletter that comes out every couple of weeks just to keep abreast of the latest change in OS 360. The proliferation of programming languages requires us to keep up with new requirements and learn new languages. Standards of documentation that we must become familiar with are being developed. Why should the programmer be asked to do the documentation? You need very little information from the programmer to generate a lot of documentation. I suggest the development of another specialty within the field of computers, that of the documentation specialist. He would work from the very beginning with the analyst and with the specifications and start documenting at that time. He would talk to all the programmers in a large-scale system to get the information at each stage of the design and implementation of a system. Finally, at the end of his programming task, he would generate the manuals and documentation needed to reach different audiences, like the user, the analyst, and the operator. Would you comment on this suggestion please?

PANEL MEMBER: I think we have moved into a specialized mode of operation in the software houses that are building and selling packages. These people do have documentation specialists. I have never worked for them, but I do know enough people who have to know that they are developing and have developed documentation specialists.

MEMBER OF THE AUDIENCE: I noticed that the documentation and its automation comes partly beforehand in the case of Dr. Rich and Dr. Mills, that some of it is ongoing documentation, and that there seems to be some after-the-fact documentation in forms like AUTOFLOW programs. But it is better to document a program before it is written. I wonder what value these after-the-fact automatic flowcharts have, or whether they are programs that were written because it is possible to write them.

PANEL MEMBER: When you analyze source code, you analyze what the machine is actually seeing. When you look at flowcharts that a programmer has written beforehand, you may or may not be looking at what the machine is seeing.

PANEL MEMBER: I think from a practical point of view you have to combine the two. I think the usefulness of any automatic flowchart technique is that you can combine your planning flowchart and make it into something that programmers will not mind updating. This motivates them to keep your planning document up to date so that you do not get into the situation of the flowchart not representing the code.

PANEL MEMBER: The assumption that there is a flowchart to start with is basically false, I think, from most of the programs I have seen. As I indicated, AUTOFLOW or any automated flowchart will show exactly what is there. That is the first time you get to see what logic was really implemented, as opposed to what the programmer wished or thought he had implemented. If you did not need it for documentation at all, it would still be very valuable as a checkout tool for the program.

MEMBER OF THE AUDIENCE: I would like to make just one comment about AUTOFLOW. We have several modes of operation. One mode is the preimplementation mode, used before you write your program. We do that internally. We draw manual high-level flowcharts. We then get them keypunched through a documentation specialist who assigns track codes. We then program. After we program and assemble, we can get several levels of documentation. We can get the logical flow in detail, which reflects the comments, the instructions, or the flowchart of the instructions, which shows the actual micrologic, a good debugging aid.

Now, if you plan your documentation before you implement your system, the programs are written, and the flowcharts really are marginal. With AUTOFLOW, however, you get high-level, detailed flowcharts.

I really feel that programmers have no real excuse for documentation problems. First-generation computers had more programming aids than computers now have. As a matter of fact, I think program productivity has fallen off, while the computers have gotten faster. If the programming aids had kept up with the hardware improvements, I think programs would now be developed more easily, and I think the problems of documentation would be less. You would get much more information from the machine.

Finally, the analysis of a FORTRAN program to produce a flowchart is very similar to the analysis of the compiler. Had these things been combined, there would have been a lot less machine time and a lot more use of automatic systems.

MEMBER OF THE AUDIENCE: I think the observation about more programming aids being used in the first generation than now was a very penetrating one and points up a very interesting fact. The process of designing and building the first generation of computers was a very unstructured one. The machines built then were extraordinarily simple. Now they are extremely sophisticated and yet are designed and built semiautomatically, almost completely automatically in this generation. What is more, when they are tested, highly automated and very sophisticated ways of checking them out, both in terms of system design and in terms of whether the indivudual machine is working, are used.

The exactly opposite process has been at work in the software business. There is obviously something wrong, and I think it must be the management of the software. I do not think it can be just the failure of individual programmers because hardware engineers are no less individual and undisciplined than programmers. But somehow or other, hardware engineers have been organized into groups, and results have been achieved that have not yet been achieved in software.

PANEL MEMBER: Let me illustrate what I mean by saying that I think we are 20 years young. There are essentially no mathematical theorems about programming itself. In the past few years some mathematical theorems about programming have been developed that are going to revolutionize the practice of programming. For example, a theorem now exists that states any program can be constructed out of tests and loops, and so on, with simply three standard figures. This corresponds to theorems in algebra that say any circuit can be designed in terms of AND/OR and NOT gates. Until programming has this kind of theoretical basis, teaching people how to program is very difficult. Now programmers are told to do it instinctively. A theoretical base is bound to come, and I think it is going to revolutionize programming. I think it will take several years, but I am trying to point out that we are just in our infancy as far as knowing how to program with a theoretical foundation.

MEMBER OF THE AUDIENCE: I feel that we do have some of that basis already; for example, the algebra that has been used in designing hardware. If we hang on to a larger entity of a program, if we can identify these, perhaps we can put them together in a more sensible manner, such as you are proposing.

PANEL MEMBER: I am just trying to state that people are now beginning to be concerned with the programming construction process itself and with the subject of writing code.

MEMBER OF THE AUDIENCE: I think there must have been a documentation problem in going from one system to the other in the diagnostic area because we have fewer diagnostics when a system comes out, and I think this is the impact in the applications programming area that we are talking about now.

PANEL MEMBER: I think that programming will change from an art into more of a science, in which programs that are correct almost all the time will be written.

MEMBER OF THE AUDIENCE: We have talked about standards, and AND/OR gates were mentioned. I think there are some basic standards, for instance COBOL and FORTRAN, which limit the language and what you can write. The syntax is varied. However, when you look at AND/OR gates you might look at the 350 or 400 IBM instructions, which let you do one particular function an infinite number of ways, which I think is a mistake. I think in the area of assembly language there are too many instructions, which then proliferate the combination of things you can do and the ways you can do it. But there are standards for COBOL, FORTRAN, and PL/I. It is just that they can be combined in different ways, just as adders and resistors can be combined. So I do not think that the problem of standards is really a problem when we talk about writing the program. The problem is how you structure and build a program, just as it is how you build hardware.

Session III

N73-19215

COSMIC PROGRAM DOCUMENTATION EXPERIENCE

Martha C. Kalar
University of Georgia

As indicated by the title, this paper deals with the experience of the Computer Software Management and Information Center (COSMIC) in computer program documentation. The first part of this paper will be a brief history of COSMIC as it relates to the handling of program documentation; the second part will summarize the items that seem to be essential for good program documentation.

On July 1, 1966, the University of Georgia was awarded a contract by NASA to receive computer software developed by NASA and its contractors and to supply copies of such material, on request, to all interested domestic parties through COSMIC. Originally COSMIC was to have been a clearinghouse type of operation; i.e., it would send to the requester a copy of exactly what was submitted. No checks were made on either the documentation or the program. This type of operation led to a number of dissatisfied customers.

In order to insure that the user received adequate documentation and a complete, workable computer program at a minimum cost, COSMIC established documentation and program checkout procedures. Time and experience have brought about changes to the original procedure.

COSMIC, today, is composed of 18 employees, 12 of whom are professionals familiar with electronic data processing and hold degrees in a variety of fields, and understand the disciplines to which the programs apply.

The professional staff is divided into two groups, one concerned with the evaluation of the documentation and one concerned with the checkout of the submitted computer program. The evaluation staff checks the documentation for completeness of vital material and assigns a class code to the document. The amount of detail, the complexity of the program, and the uniqueness of the solution all play a part in determining which class code is assigned to these programs. The programmer staff performs a check on each program submitted to the library to determine whether all nonstandard routines are present in the program deck. There are four machine types available to our programmers, the IBM 360, the IBM 7094, the CDC 6400 at the University of Georgia, and the UNIVAC 1108 at the Georgia Institute of Technology. Programs written for any one of these machines are compiled before dissemination; however, programs written for other machines must be assumed to be executable when they are disseminated.

Of some 2800 program packages submitted to COSMIC, 60 percent have been rejected for one reason or another by either the programmer or the evaluation staff. The poor

Preceding page blank

quality of the documentation received is a major factor in the rejection of the program package. Many times illegible documentation has been received, and the program has therefore been rejected. Programs have also been rejected because they are too short or too special purpose to have any value to organizations other than the originator's. Other submitted packages have not contained vital segments of the documentation, making them unusable. For example, COSMIC has received documentation that was a Xerox copy of a listing, with penciled notes on the sides. Documentation of this caliber cannot be disseminated.

COSMIC has encountered a variety of problems in the content of the documentation submitted. Experience has shown that the problem is most often in the user instructions. It is assumed that a purchaser of a COSMIC program is buying the program because it will solve his problem directly or because it can be modified slightly to solve his problem. Therefore, the user knows most of the technology involved or is at least familiar with its purpose. The reason the user needs the program is to obtain the desired results without having to write the program himself. The user, therefore, needs detailed user instructions that are easy to follow. The following is an example of poor user instructions: A Xerox copy of the handwritten instructions, "Use standard IBM OS/360 job control setup," was submitted as documentation. Needless to say, the documentation was rejected. Complete instructions would have contained a listing of a sample deck setup and samples of input and output format. These are needed because machine configurations differ and what is standard to one installation may not be to another. The input and output formats are needed so the user can test his results and knows what to expect of his output appearance.

Because of deadlines and overlapping projects, documentation does not always receive its fair share of the time allotted for these projects. When one works closely with a program for a period of time, certain terminology and concepts become very familiar, and when the documentation is updated, these terms and concepts might be omitted or overlooked. The potential user of the program, however, most likely will not know its routine terminology and familiar concepts; therefore, problems arise. The programmer should be aware of his users and should gear his documentation toward the novice, the user who knows very little, if anything, about the program.

COSMIC's purpose is to disseminate programs that any potential user can employ. Certain areas of documentation are essential and shall be outlined here:

(1) Program name (official name, acronym, and program title)
(2) Identification number (NASA, contractor, or other number; COSMIC references programs in our library by the NASA-assigned "flash-sheet" number)
(3) Installation name (name and location of the center where the program was developed)
(4) Date (date which program was completed)
(5) Author(s) and affiliation(s) (The author of the program is usually the person who does the actual programming and design work. If these tasks are separate, both names should be given.)
(6) Language (the programming language in which the program was written)
(7) Computer or machine requirements (computer, minimum configuration, level of compiler, and other requirements for the execution of the program)

(8) Functional abstract (approximately 300 words) including the following:

   (a) Description of the program (The problem that the program is designed to solve should be presented in such a way that the reader may identify elements that are analogous to his own problem.)
   (b) Method of solution (When the method is well known or documented in standard publications, it should be identified by reference. Modifications to well-known methods, new methods, or novel combinations of methods should be fully described to indicate their applicability.)
   (c) Special features of the program (Processing features and options that contribute to the uniqueness of the program should be summarized. Types of input and output should be discussed in terms of their potential value in solutions of problems.)

(9) User instructions

   (a) Input preparation formats and options (precise definition of all variables, exact format and arrangement of input parameters, required card or tape format for all input data, and sequence of control statements)
   (b) Output formats and options (These should clearly explain all output variables; some note regarding accuracy of results also should be included.)
   (c) Data restrictions (The user should be provided with a full explanation of any data restrictions such as those constituting illegal input, numerical or data-set limitations, and the number of or size of the data sets that can be handled by the program.)
   (d) Procedural references (manuals and detailed documentation required to use the program)

(10) Sample input and output models

The documentation that COSMIC receives, in most cases, does not include all these items. Standards at COSMIC have been minimal in the past but are constantly being upgraded. (See appendix.) If the documentation is deemed insufficient, more information is requested from the originating center. If more information is not available, the program must be rejected. On some programs, this is all that can be done. The turnover among programmers is fantastic. A programmer remaining at one job for 2 years many times will have seniority in a department. Therefore, contacting the originator becomes a difficult task. But on the programs being written now, we hope to establish standards to obtain complete documentation initially with as much information as possible in order to anticipate later questions.

APPENDIX—COSMIC DOCUMENTATION AND PROGRAM STANDARDS HANDBOOK

I. INTRODUCTION

COSMIC (COmputer Software Management and Information Center) was established to evaluate computer software developed by governmental agencies and then disseminate the evaluated submittals to other governmental agencies, as well as industrial, educational, and research institutions. To expedite the technical aspects of this process, it is necessary for COSMIC to receive properly prepared documentation and program packages from submitting field centers and contractors. To explicitly state COSMIC's requirements for submittal packages is the primary purpose of this handbook.

COSMIC is cognizant that all documentation packages received will not meet the exact format as outlined in this pamphlet; however, it is imperative that all information requested herein be included with the package regardless of the format chosen.

It is anticipated that this volume will—

(1) establish a much needed and easily implementable standard for documentation;
(2) clarify the definition of a complete program deck;
(3) promote a better understanding among all offices and agencies involved; and thus,
(4) increase the efficiency and effectiveness of the entire project.

II. DOCUMENTATION CRITERIA

A. General

Documentation which meets the COSMIC standards must include the amount of information necessary to inform a prospective user of the precise problem which the computer program is designed to solve and to enable a qualified programmer to input the required data, successfully run the program, and obtain the desired results. Below is a chart of documentation criteria, each of which will be defined in the following text.

DOCUMENTATION CRITERIA

SPECIFIC REQUIREMENTS

1. Description of the Problem
2. Method of Solution
3. Program Language
4. Machine Requirements
5. User Instructions
6. Operating Instructions

OPTIONAL REQUIREMENTS

1. Program Timing
2. Accuracy of Results
3. Sample Input and Output
4. Flowchart
5. Listing

B. Specific Requirements

The following information must be included in the documentation for it to meet the COSMIC standards:

1. Description of the Problem–The description must include a **complete** definition of the problem which the program solves. The thoroughness and sophistication of this definition is determined by the sophistication and degree of difficulty of the problem itself. For instance, a simple mathematical routine may be described in one sentence, whereas a description of a program designed to construct electronic printed circuit boards may require a multiple number of pages.

2. Method of Solution–This requirement must include the programming techniques or methods used, supporting theory, design, and computational equations with their derivations to substantiate or illustrate the program.

3. Program Language–A statement of program language must include all levels of languages found in the submitted deck (e.g., FORTRAN IV, MAP, OBJECT) as well as the compiler necessary to process the languages.

4. Machine Requirements–An explanation of machine requirements must encompass not only the computer system for which the program was developed but also all peripheral equipment utilized by the program (e.g., disks, drums, consoles, tape units, display devices, plotters, etc.). Also mandatory is the level of the operating system on which the program executed (e.g., IBM-360/65, Release 14; CDC-6600, Scope 3.1; etc.) as well as the amount of core a program occupies once loaded.

5. User Instructions

   a. Input Instructions–These instructions must provide the user with the information necessary to prepare his data for input to the program. They should include:

      (1) precise definition of all variables;
      (2) the exact format and arrangement of all input parameters (object time variables); and
      (3) the required card or tape format for all input data to be processed. It must be noted if the input requirement is for a specialized format, e.g., NASA formatted telemetry tapes.

   b. Output Requirements–The user instructions must also contain a description of the output data formats and types of output devices; e.g., card punch, printer, magnetic tape, etc. In addition, the instructions must include an example to illustrate both the input deck setup and the corresponding output.

   c. Data Restrictions–The assumption must be made that the user knows nothing of the mechanics of the program; therefore, any data restrictions or illegal input should be specified. For example:

      (1) x cannot equal zero;
      (2) y must be less than 200;
      (3) x cannot equal 5 unless y is less than 4.

   d. Program Structure–A list of all decks in the program, the main program as well as any subroutines called, must be included. If a routine is to be included in more than one subsection (e.g., chain, overlay, etc.) of a program, please so indicate.

6. Operating Instructions–This information must provide the computer operator with step-by-step instructions pertinent to the execution of a program. It must include:

   a. tape assignments or selection (Designate tapes required for input, working, and output for successive runs.);

b. deck setup;
c. control and sequencing information; and
d. special controls and requisite operator actions (e.g., console instructions).

C. Optional Requirements

The following information, although not essential, will facilitate processing and use of a program.

1. Program Timing—Timing information should include the computer time required for a run with a certain number of data points or check points, or the computer time required for an average run.
2. Accuracy of Results—This section should include the number of decimal points or number of significant digits which can be expected in the answer. Where some inputs are based on sampling, both the accuracy of the estimates and the reliability of the output should be supplied.
3. Sample Input and Output—A description of a sample problem, an example of the input data required to run the program, and resulting output from a run of the respective input should be included.
4. Flowchart—This must be a structural flowchart of the sequential logic and decision points included in the program. Machine-produced flowcharts of the exact programming techniques cannot be used to satisfy this requirement as they merely amount to a listing of the programs and do not briefly and concisely reflect the inherent logical flow of decisions.
5. Listing—This must be a post-list of the assembled program submitted to COSMIC to be used as an in-house aid in processing programs.

III. PROGRAM CRITERIA

A. Card Deck and Tape Submittal Formats

Following is a list of requirements compiled by COSMIC in an attempt to standardize program handling processes and to eliminate misidentification of submitted programs:

1. Card Deck Submittals—These must be clearly marked with the respective program identification numbers.

*2. Tape Submittals—It is requested that 7-track tapes be used. If this is impossible, 9-track will be accepted.

   a. Tapes must be recorded:

      (1) at 556 or 800 bpi,
      (2) in unblocked card image format (84 characters per record for BCD or 168 characters per record for binary),
      (3) with a complete program package (main deck, subroutines, data, etc.) in the same file,
      (4) with each complete program package separated by an End-of-File card (blank except for a 7-8 multiple punch in column 1),
      (5) with multiple 7-8 cards following the final program on tape.

   b. Programs must be identified by number, title, and file position sequence on tape. This may be accomplished with a cover letter or a label on the tape reel.

Note added in proof: These conventions have been revised in line with improved computer technology. The conventions stated here are not presently in use.

B. Definition of a Complete Program

An explanation of COSMIC's definition of a complete program is pertinent at this point. To be considered complete, a program must include:

1. main program;
2. all non-standard (not included with operating system as normally installed by manufacturer) subroutines called within the main program or by other subroutines in the package; and
3. all plotting routines called (If this is impossible for proprietary reasons, submit a dummy subroutine deck with all user called entry points; also, include with the documentation complete input and output variable formats for the routines used.).

C. Mode of Submittal Programs

It is imperative that COSMIC receive source decks rather than object mode decks. It is seldom that a disseminated program can be implemented by a purchaser without modifications being necessary. To facilitate modifications and, thus, wider usability of COSMIC programs, we publish only source programs.

DISCUSSION

MEMBER OF THE AUDIENCE: I wish to raise the question of standards versus guidelines. My understanding is that standards are something required, and guidelines are something to be desired. It seems to me that if documentation standards are insisted upon, many programmers will simply refuse to adhere to them.

KALAR: If most users can use documentation in a certain form, then I think the best thing to do is to try to put it in that form. If the form is pretty well agreed upon, then I think that people ought to try to conform to it. Call it standards or guidelines, I cannot determine between the two. I do not think you can enforce anything.

MEMBER OF THE AUDIENCE: I would like to try to answer that. I do believe that some minimum amount of information should be available to a potential user so that he can make some choice as to whether he wants to make a substantial investment in some of the documentation, which may run into thousands of dollars. I do believe that a standard or a standard requirement or specification may be needed in this area.

MEMBER OF THE AUDIENCE: You have given us a list of things that you desire to see in documentation. Has this been disseminated to your customers?

KALAR: A partial list is in the appendix of my paper. This is a little bit different from the one that COSMIC is now disseminating to its customers.

MEMBER OF THE AUDIENCE: Is it a regular procedure to advise the customers or the people that send you programs of the problems that you see as you go along?

KALAR: Generally, most of these items are covered in the appendix. When programs are submitted, if they are deficient in a certain area, we will tell the senders what areas to send us as documentation. These exact items are not written down yet, but they should be within the next couple of months.

MEMBER OF THE AUDIENCE: To your knowledge, has COSMIC had an opportunity to review the proposed NASA NHB standards on documentation?

KALAR: I do not know.

MEMBER OF THE AUDIENCE: How do you determine the costs for the distribution of programs?

KALAR: By the number of cards in the deck for the programming. The documentation is 10 cents a page. An average cost is about $275 per program.

MEMBER OF THE AUDIENCE: You said you like to disseminate programs that any individual can employ. Well, I question this. We have taken the other tack in the nuclear field. We have said we want to disseminate programs to installations that have people competent in both computer science and nuclear science because we feel that if you really disseminate programs to anyone, you can spend a fortune trying to train them.

KALAR: I do not think we can be so choosy about our customers. Whoever wants to buy a program can buy one, and if they can understand it, they can use it.

MEMBER OF THE AUDIENCE: It seems to me you have to do a lot more work. More documentation is needed, and these people must be brought in and trained how to use these programs.

KALAR: Most of our programs now being disseminated are accounting-type programs, programs that the small businessman can use without any extensive knowledge.

MEMBER OF THE AUDIENCE: I have a question about your organization. How did a university get into disseminating programs that industry has paid fortunes to get?

MEMBER OF THE AUDIENCE: Could I answer that question? I am the COSMIC specialist at Goddard Space Flight Center with the Technology Utilization Office. COSMIC is mainly a nonprofit institution. NASA has a duty to distribute the technology that NASA develops to people in the public sector, that is, commercial, profit, and nonprofit organizations that may have a need or a desire to use any part of the technology that we develop. Computer programs are considered a part of that technology. COSMIC's function is to distribute those programs to those in the general public who may find them useful, thereby increasing the productivity and welfare of the general public. There is no profit involved to COSMIC. The programs that industry develops under NASA contracts belong to the Government. What the Government does in this instance is make that property available to the commonweal. I hope that answers your question.

MEMBER OF THE AUDIENCE: What is the general turnover in programs and purchases at COSMIC?

KALAR: I think we sell around 60 or 80 packages per month and receive probably an average of 50.

MEMBER OF THE AUDIENCE: One problem with documentation is that we may meet the documentation requirements for a Government contract. Then the contract monitor says to submit it to COSMIC. We submit it to COSMIC and receive a different set of documentation requirements. We go back to the contracting officer and ask for the money to document the program or the system for COSMIC, but they refuse. In other words, who is going to pay for documentation?

MEMBER OF THE AUDIENCE: This is one of the things that falls in my area. I believe that most NASA software documentation requirements now incorporate the COSMIC requirements for program documentation. What often happens is that the contractor does not regard these as essential, because in the past the documentation specifications really have not been enforced. We now demand that these requirements be met.

MEMBER OF THE AUDIENCE: Do you review the request for proposal (RFP) to see that the requirements . . .

MEMBER OF THE AUDIENCE: I do see some of them, but I believe that most of our RFP's for documentation now include those requirements.

N73-19210

AUTOMATED DOCUMENTATION OF AN ASSEMBLY PROGRAM

Valerie L. Thomas
NASA Goddard Space Flight Center

The programmer's burden is greatly reduced if he is willing to invest some time in setting up his program so that it can be used as input to a program which will automatically document it. This paper will present ideas of how this could be implemented for an assembly language program; the examples will be from a META-SYMBOL program which is run on the XDS 930 computer. For the purpose of this paper, the documentation program will be called DOCMNT.

Before the characteristics of DOCMNT can be explained, some format rules must be established for the assembly language program so that DOCMNT can scan the input program (see fig. 1) and find all the information that is necessary for the documentation. The program should be divided into four major parts: identification, main routine, subroutines, and data.

The identification should consist of comment cards having asterisks in the first column and in the second column information pertaining to the program, such as, the name of the program, the programmer, and the operating system. DOCMNT scans these card images looking for the following format:

```
*    PROGRAM–OFOINT
*    PROGRAMMER–VALERIE L. THOMAS
*    OPERATING SYSTEM–MULTISATELLITE OPERATING SYSTEM (MOS)
*    MACHINE AND LANGUAGE–XDS 930, META-SYMBOL
*    DATE–JULY 14, 1970
```

DOCMNT gets OFOINT, VALERIE L. THOMAS, etc., and stores this information in the appropriate place in the documentation. (See fig. 2 for the format of the documentation.) DOCMNT would have the capability of storing OFOINT MAIN beside ROUTINE and of storing for each of the subroutines of the program, the subroutine name beside the word SUBROUTINE; for example:

Main routine		*Subroutine*	
ROUTINE:	OFOINT MAIN	SUBROUTINE:	COMP
PROGRAM:	OFOINT	PROGRAM:	OFOINT

The second major part of the program is the main routine. This should be preceded by some comment cards, one of which contains the words FUNCTIONAL DESCRIPTION and is followed by the description of the main routine; for example:

PRECEDING PAGE BLANK NOT FILMED

```
*       PROGRAM-OFOINT
*       PROGRAMMER-VALERIE L. THOMAS
*       OPERATING SYSTEM-MULTI-SATELLITE OPERATING SYSTEM (MOS)
*       MACHINE AND LANGUAGE-XDS 930, METASYMBOL
*       DATE-JULY 14, 1970
*       FUNCTIONAL DESCRIPTION-OFOINT PROMPTS
*          THE OPERATOR AND USES THE ANSWERS
*          TO UPDATE THE PROMPT TABLE.

STRT    PZE
        LDX     MOSINK
        LDA     1,2
        STA     COMPU
        MIN     STRT+2
        MIN     STRT+3
        MIN     GTCTR
        LDA     GTCTR
        SKG     =30
        BRU     STRT+2
PROMTS  BRM     COMP
        BRM     CHNREQ     COMPUTER
                           CHANNEL REQUEST

        LDA     TABCTR
        SKE     =506
        BRU     LDCA-1
        BRM     MOSXIT

*       FUNCTIONAL DESCRIPTION-COMP DETERMINES WHICH COMPUTER
*       IS BEING USED. DEFAULT IS COMPUTER 1.

COMP    PZE
        LDA     =0
        STA     PRMNO
        LDX     =-20
        LDA     PROTS,2
        BRM     PRNTT
        BRM     MOSREQ
CM1     PZE     TYPOT
        PZE     RESERVE
        PZE     =20
        .
        .
        .
        .
        .
        .
        BRU     CM
*       ERROR CONDITIONS-COMP IS NOT EQUAL
*          TO 1 OR 2. "MESSAGE GARBLED"
*          IS TYPED OUT.
        BRM     GARB
        LDX     PRMNO
        BRU     PROMTS,2
        BRR     COMP

*
```

Figure 1.–Sample program.

```
         .
         .
         .
*        DATA
DISA     DATA   044160000
DISB     DATA   012160000
TIME     RES    2

         .
         .
         .
*        DATA   FORMAT-THE PROMPT TABLE
COMPU    DATA   01          1
CHANL    DATA   022         2
WALL     DATA   045         3
*        DATA   FORMAT-THE CALIBRATION TABLE
A4L1M    DATA   1,0,51
         DATA   0,51
A5L1M    DATA   2,0,0377
*        DATA   0,128,255

         .
         .
         .
         END
```

Figure 1 (continued).–Sample program.

```
*    FUNCTIONAL DESCRIPTION–OFOINT PROMPTS
*      THE OPERATOR AND USES THE ANSWERS
*      TO UPDATE THE PROMPT TABLE
```

DOCMNT continues to scan the program and picks up this information and stores it in the documentation under the heading FUNCTIONAL DESCRIPTION.

DOCMNT will follow the same format for documenting the main routine and each of the subroutines. The argument to the subroutine and the output from the subroutine are assumed to be in the A register. If there is more than one argument, DOCMNT will check for a comment card containing a key (OP1, OP2, OP3, or OP4) to locate the argument list; for example:

```
*  OP1
*  OP2
```

When the key is found, the following information is known:

OP1	A calling sequence was used.
OP2	The argument list follows the instruction (branch to the subroutine).
OP3	There is a pointer to the argument list in the A register.
OP4	The programmer must insert additional comment cards to explain where the input and output arguments are located.

IDENTIFICATION

ROUTINE–OFOINT   MAIN
PROGRAM–OFOINT
OPERATING SYSTEM–MULTI-SATELLITE OPERATING SYSTEM (MOS)
PROGRAMMER–VALERIE L. THOMAS
MACHINE AND LANGUAGE–XDS 930, META-SYMBOL
DATE–JULY 14, 1970

ARGUMENT LIST: NONE

FUNCTIONAL DESCRIPTION: OFOINT PROMPTS THE OPERATOR AND USES THE ANSWERS TO UPDATE THE PROMPT TABLE.

METHOD

(1) THE PROMPT TABLE IS PICKED UP FROM OFORUN.
(2) A SUBROUTINE IS EXECUTED FOR EACH PROMPT AND THE PROMPT TABLE IS UPDATED.

READ/WRITE

R/W	DATA AREA OR DEVICE	CHARACTERS/WORD
NA	NA	NA

REQUIREMENTS

SUBROUTINES USED: COMP, CHNREQ
MEMORY: 20 OCTAL WORDS
RUNNING TIME: 30+ CYCLES

ERROR CONDITIONS: NONE

SUPPORTING INFORMATION

THE PROMPTS ARE PICKED UP FROM THE PROMPT TABLE IN OFORUN, UPDATED, AND STORED BACK IN THE PROMPT TABLE IN OFORUN. THIS ALLOWS THE PROGRAMMER TO SAVE THE PROMPT ANSWERS ONCE THEY HAVE BEEN CHANGED FROM THE DEFAULT VALUES.

IRREGULAR RETURNS FROM SUBROUTINES: NONE

DATA FORMATS

THE PROMPT TABLE

```
COMPU   DATA   01    1
CHANL   DATA   022   2
WALL    DATA   045   3
```

THE CALIBRATION TABLE

```
A4L1M   DATA   1,0,51
        DATA     0,51
A5L1M   DATA   2,0,0377
        DATA   0,128,255
```

EXTERNAL REFERENCES

MOSLNK
MOSXIT

Figure 2.–Automatic documentation format.

The information pertaining to the arguments goes under the heading ARGUMENT LIST. DOCMNT scans the routine for branches outside of the routine and puts the information under the heading IRREGULAR RETURNS FROM SUBROUTINES. DOCMNT stores the identification information and the functional description. It then goes through the remainder of the routine to gather more information for the documentation. It counts the instructions as they are investigated to provide the octal count of the number of words of memory used. It has a table which it consults to determine the minimum number of cycles used for the running time. If the routine contains a loop, an unconditional branch, or a branch to a subroutine, the octal word count will have a plus sign next to it, denoting that the minimum time is calculated. DOCMNT searches the routine for the branches to subroutines and lists the subroutine names beside the heading SUBROUTINES USED. If any input/output is used in the routine, a word can be found which gives information pertaining to the device used, whether it is a read or write, and the number of characters per word. This is stored under the heading READ/WRITE. If the routine contains an error routine, the branch to the error routine should be preceded by the following:

```
*     ERROR CONDITIONS—COMP IS NOT EQUAL
*        TO 1 OR 2. "MESSAGE GARBLED"
*.       IS TYPED OUT.
```

DOCMNT scans for error conditions and stores the information that follows in the documentation under ERROR CONDITIONS. DOCMNT also searches for external references, which are listed under the appropriate heading.

The third major part of the program is the group of subroutines each of which is preceded by comment cards (containing the functional description card) and followed by at least one card with an asterisk in the first column.

The last major part of the program is the section containing the data. This section is preceded by the following card:

```
*     DATA
```

To put some sections or tables from the data under the heading DATA FORMATS (in the documentation), a data format card with comments is inserted ahead of the data and an asterisk card after the data; for example:

```
*   DATA FORMAT—THE PROMPT TABLE
COMPU     DATA    01     1
CHANL     DATA    022    2
WALL      DATA    045    3
*
```

The information for the headings METHOD, SUPPORTING INFORMATION, and EXTERNAL REFERENCES is obtained from card input. The cards in the deck containing the external references can be read in and the information stored into a table of external references until DOCMNT is ready to use it. The cards containing the supporting data can be read in and stored in a table. Because every routine will have cards in the method deck, it is best to read in the cards for each routine as it is being used by DOCMNT.

Flowcharts for each routine are done by AUTOFLOW.

After the documentation is completed, a table of contents can be printed out.

DISCUSSION

MEMBER OF THE AUDIENCE: How do you get the running time for the subroutine?

THOMAS: Each instruction has a certain number of cycles, and we can get an estimate of the minimum time by counting out the number of cycles per instruction. If there is a branch to a routine we take the minimum time, which would be just the instruction and the branches to the routine. If there is a loop, we will just take each instruction within a loop. So if we have a case like that we will put beside the running time a plus, which indicates that it is the minimum time.

MEMBER OF THE AUDIENCE: Do you have anything in the system that forces the programmer to provide the information that you need?

THOMAS: No. At present, this is a theoretical system. But if you do not put it in, you will not get out what you want. If you want something that is detailed, you will put in what is necessary. These keywords can be inserted after the program is written, but this should be done before for a very well-documented program listing besides having the automated documentation.

MEMBER OF THE AUDIENCE: Are there plans to get the system built and to get it operational?

MEMBER OF THE AUDIENCE: For what it is worth, there is a system similar to this in operation in a large industrial organization. It is used as a prelist and a requirement, and it works very well. In other words, the programmers must comply. They have no great turnover because of this, particularly these days. The system actually is not used on keyboard. It is based on COBOL, and it does go through, picks out all comments, notes, and paragraphs; sets it up; and does an absolutely excellent job. They also take this and automate it to a quick-input index, which is distributed worldwide and is used as a catalog library similar to that described by COSMIC.

N73-19217

AUTODOCUMENTATION

Jay Arnold
Computing & Software, Inc.

Currently available automated documentation systems, like the ones described at this symposium, perform the functions for which they were intended quite admirably for the most part. They have proven to be useful tools in the area of retrospective archival documentation and as aids in finding logic problems. However, several documentation needs remain that current systems do not fulfill.

As yet, systems that can recognize a program by type and categorize the process or that can describe the application for which a program was intended have not yet been developed. But both of these functions are necessary if automated documentation systems are to be used to solve some of the serious problems facing this industry.

It is fairly obvious that the industry is plagued with "specialization" and "originality" syndromes, that routine programs are written and rewritten for each new application. Lack of adequate documentation for the vast store of existing programs only serves to further aggravate these problems.

To decide whether an existing program can be used for a new application, it is helpful to know both how the program was originally used and what functional processes are contained in the program. The latter is particularly important in interdisciplinary transfers. But few programs have adequate documentation of this type. Most programmers preparing individual documentation are unable to see how their program or segments of it may be used in other areas. Therefore, the cost of determining the capabilities of programs or program segments usually precludes their use and forces the development of additional programs.

Automated documentation systems, with expanded capabilities, would provide a means of reusing existing programs by allowing a relatively inexpensive determination of program capabilities. This paper will present some comments on an approach to such a system. These comments do not describe any existing system and are presented solely with the intent of stimulating thought in the area.

The approach stems from observing human analysis of programs. Of course, developing a machine system on the basis of a human approach is not always the best or an efficient technique. However, it appears to be one approach to the problem.

Most programmers conduct an analysis using a source listing of the program, a description of the program input, and a description of the program output or, when available, a sample of the output itself. They tend to begin their analysis by studying the sample output to determine what they can of the original intent of the program and to look at the output data elements to provide them a "link" into the program.

Next, they take up the source listing to find an output statement that corresponds to the data element of interest from the output. From this point, their analysis is aimed at determining the content of the program rather than its original intent to determine whether the methods employed in the program are of use to them.

The human approach, then, usually starts with the program output, does an analysis of original intent, uses the output as the entrance to the program, and then proceeds to a content analysis. It should be possible to develop an automated system based on this approach. Basically, such a system would be an output-to-input analysis as opposed to the more common input-to-output flow analyses.

There are two general types of program output with which an automated system would have to contend, print and nonprint. Since the former presents less problems, it shall be considered first.

One goal in designing any automated system should be the minimization of requirements with which the programmer has to conform. The use of special control cards solely for the documentation system is an undesirable constraint. Even ordinary comment cards, while highly desirable in any documentation package, should not be a requirement for an automated system. One guideline, however, could be employed with minimal limitations on the programmer. That is the use of self-descriptive labels on all printed output as well as on nonprinted output where feasible. Since labeling is a relatively common practice, it should not prove to be a severe constraint to a programmer.

Most printed output contains two broad categories of information: report description (or header information) and data description (or lable information). In a majority of cases, some form of these two information types are present on printed output.

Header information usually includes project names, data-set descriptions, experiment types, calculation methods, names, places, dates, times, and other information that describes the purpose of that particular program output. It is this part of printed output that is of the greatest use in determining the original application.

The label information, on the other hand, pertains more specifically to the data that the program generates. Row and column labels indicate information about each sequence of calculations within the program. From these data, parameters that are being calculated and the elements to which they correspond can be deduced. In many cases, much application information is available on the printed output.

Generally, the analyzer processes this information about the program by some type of semantic and syntactic analysis. In the case where only limited information is present in a nonsentence structure, it is probable that semantic analysis would be predominant. Most programmers could probably deduce quite a bit about a program in a familiar application area by noting only a few keywords on a printout because the scope of the application area also limits the meaning and context of the terms which we see. At GSFC, the acronym OGO would immediately suggest a satellite rather than a Government organization. Given enough of these terms on a printout, within a limited context, the program application should be fairly accurately described.

The programs that do not produce printed output, such as sorting routines, utilities, and math function subroutines, will now be considered. For some types of nonprinted output, the original application cannot be easily determined, but these are exactly the types of programs whose original intention is irrelevant. Since most of these cases are the general-purpose routines that can be used for almost any application, the decision as to whether they can be used in a new application does not depend on the original intent. However, some types of more specialized programs do not produce printed output where information pertaining to the original application would be beneficial. Even in the case of nonprinted output, three sources of information might be available: descriptions of output data sets, labels on the data sets, and cross-reference information available from printed data sets.

Probably the best place to look for the original application is at the descriptions of the output data sets. These descriptions, such as those contained in IBM 360 job control language, provide information about the size, type, and organization of the intended output. In many cases this provides clues to the application that could not be obtained from the program itself. A second source of information might be the labels on the data sets themselves. However, this information is not always available to the analyzer. When available, it can provide additional descriptive information about the original output intent. One last method of obtaining information about nonprinted outputs is to cross-reference it to information on a printed data set. In some cases, the application of the nonprinted output data set can be deduced from information on the printed output.

These are some of the means the analyzer has to deduce the original intent of the program. His approach to understanding the content or functioning of the program should be the next topic. Most computer programs, especially scientific ones, are, for the most part, a heterogeneous collection of calculations or data-manipulating processes applied to a problem area. Unfortunately, a program is usually considered as a single entity, rather than as the sum of its parts, which tends to distort the programmer's view of the inherent capability and usefulness of the parts of the program.

A brief look at any program will show that each output data element is produced by a unique sequence or "pattern" of calculations or processes. While it is true that many of these processes may overlap in multioutput programs and that some may be prerequisite to others, each output element can be traced back through the program to yield a unique pattern. Each of these processing sequences could, in most cases, be separated from the program and become an independent module with an identity and function of its own. Thus, the analyzer's task of determining program content is reduced to several subtasks of determining each of the patterns that yields an output. He must enter the program at each output data item to ascertain the pattern of processes that led to it.

As the analyzer traces the sequence of calculations and the pattern begins to clarify, he attempts to compare the developing pattern with those with which he is familiar. Unfortunately, the wide variation in programming techniques that can be employed to implement a particular well-defined process precludes the possibility of a simple comparative process. Each programmer may have a unique way of translating an established technique into coding. But while the latitude is wide, it is still finite. He is limited by the language syntax as to how he may code this process.

The analyzer must therefore be equipped to recognize the pattern from some finite range of pattern variations. To accomplish this, he must either be familiar with the entire

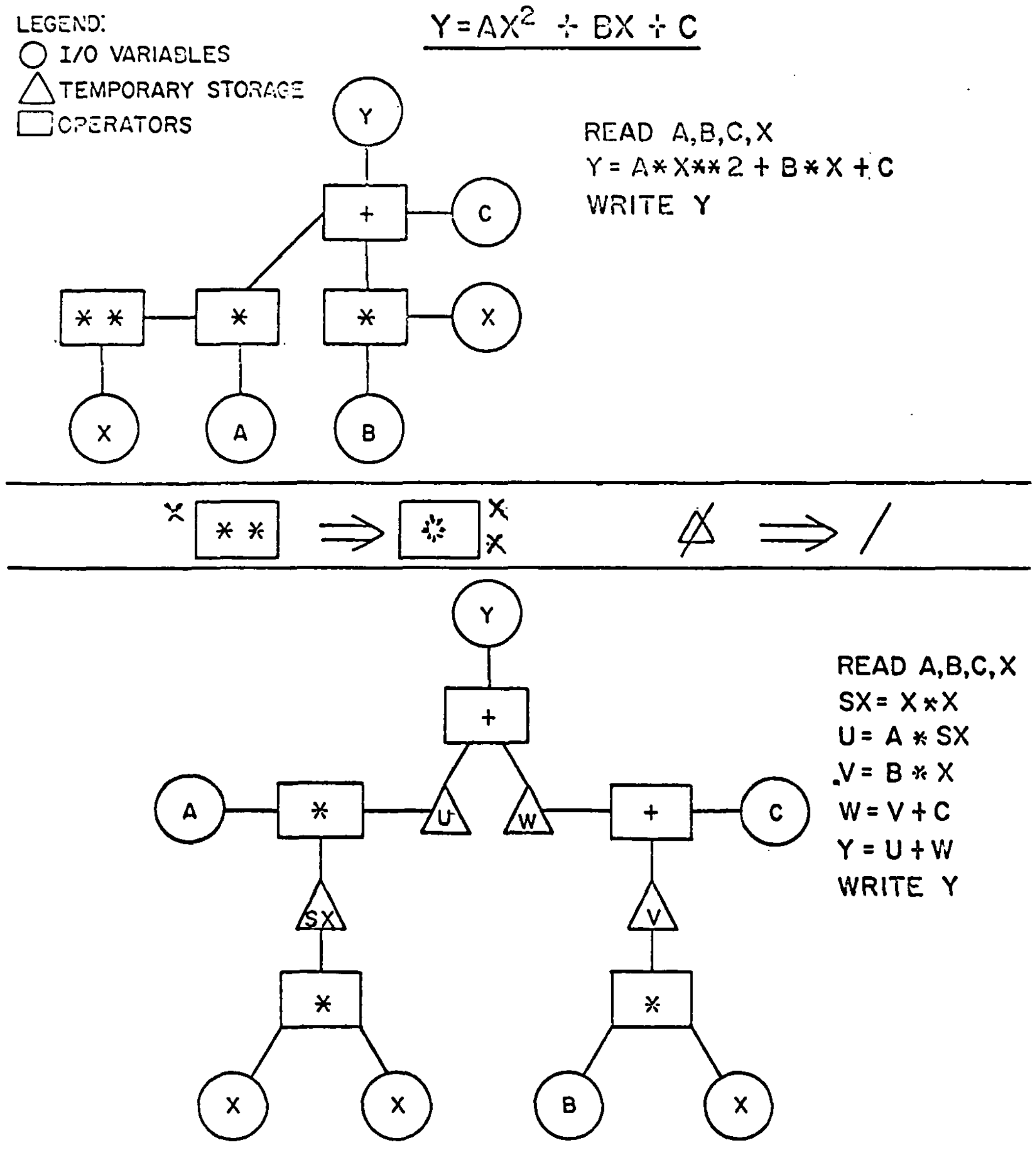


Figure 1.–Tree-structure variations.

range, which is certainly possible in many cases, or he must be capable of reducing the pattern before him to a familiar one. Whereas the latter tends to be a more difficult task, it has the advantage of requiring a familiarity with only one variation for each pattern, an advantage that might prove significant for an automated system.

While it is difficult to say exactly how a person organizes a pattern in his mind for recognition, a fair analogy might be the tree-structure representation, as indicated in figure 1. This technique has been chosen to represent patterns for two reasons: The structure can be built one level at a time, much like a human analyzer, and it is readily amenable to automated

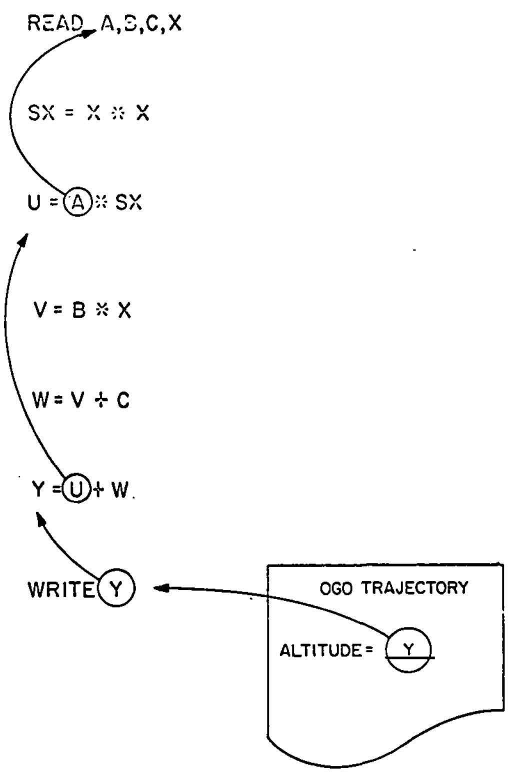


Figure 2.—Assignment statements.

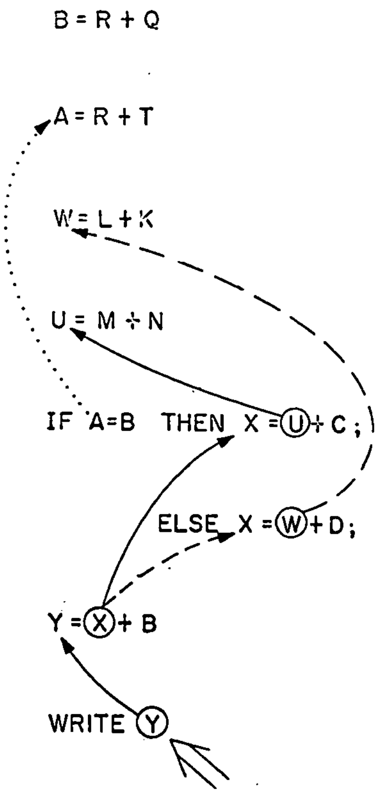


Figure 3.—Conditional assignment.

processing. As the example illustrates, the same mathematical function coded in different ways initially produces different structures. However, by applying simple techniques, the patterns can be shown to be identical.

The components of these patterns will now be considered. Although the nature of program analysis makes it, in some respects, language dependent, some of the more common general aspects can be discussed.

Once again, analysis begins with program output. The analyzer has found a data item in the output and has sought out the corresponding output statement in the program. He has identified the variable that corresponds to the data item and is about to trace the pattern of calculations.

Figure 2 shows a program consisting solely of assignment statements. The pattern is traced from the output statement to the left side of an assignment. From there, each of the preceding variables is traced to its origin, either an input or generation point. Each of the variables and operators encountered can be stored in a tree format such as the one in figure 1.

Assignment statements, though plentiful in programs, would probably be the simplest to analyze in an automated system. Although no analysis details have been worked out for the more complex processes, brief comment on some of the most common is possible.

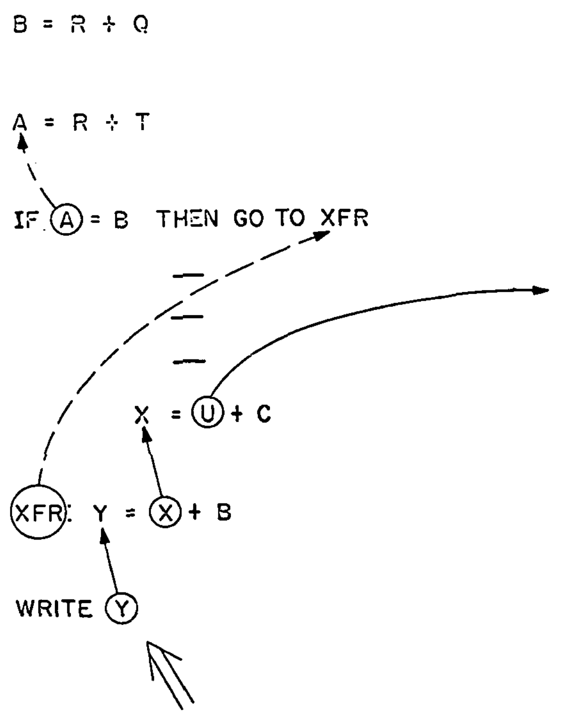


Figure 4.—Conditional branch.

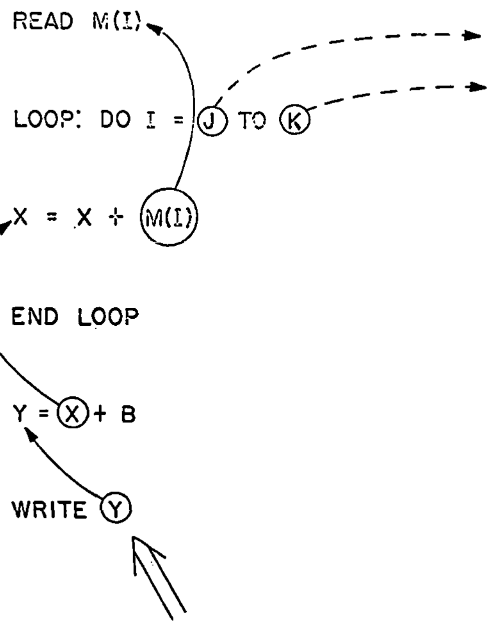


Figure 5.—Loop.

Conditional processes are probably the most common of those that might be difficult to analyze. This class of processes includes such types as conditional assignments, conditional branches, and loops.

First, the analysis of conditional assignments should be considered (fig. 3). Here one of two assignment paths can be followed depending upon the validity of the condition. The proper analysis of this statement would require tracing the pattern for both alternatives and also the pattern leading to the condition. Analysis of these three patterns would yield a complete picture of the structure of this segment. One thing this analysis might have indicated was that these were two discrete patterns, one or the other of which was selected on the basis of input data.

Next, the conditional branch shall be considered (fig. 4). In this case, the execution of an entire sequence is dependent upon the condition. If there were a direct assignment path from the output variable to the labeled statement, the presence or absence of that output would be determined by the condition. Here, analysis would require a trace of both the variable and the condition structure.

Finally, loops should be considered. In figure 5, a straight trace path is interrupted by one. To analyze this pattern, the number of iterations and nature of any discontinuities must be known, especially those near either end of the iterations. If a manual analysis were being performed, the flow of the loop at its first iteration, its second, the next to the last, and the last would be determined. Of course, if there were a conditional branch out of the loop, rather than a fixed number of iterations, that would have to be taken into consideration.

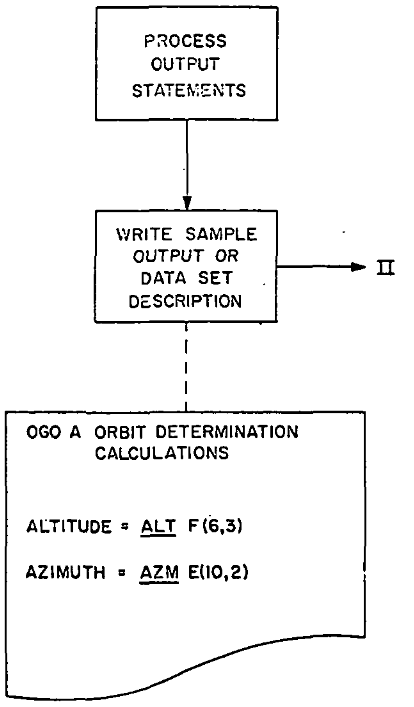

PROCESS OUTPUT STATEMENTS
WRITE SAMPLE OUTPUT OR DATA SET DESCRIPTION
II
OGO A ORBIT DETERMINATION CALCULATIONS
ALTITUDE = ALT F(6,3)
AZIMUTH = AZM E(10,2)

Figure 6.—Step 1 of an automated analyzer: Sample output.

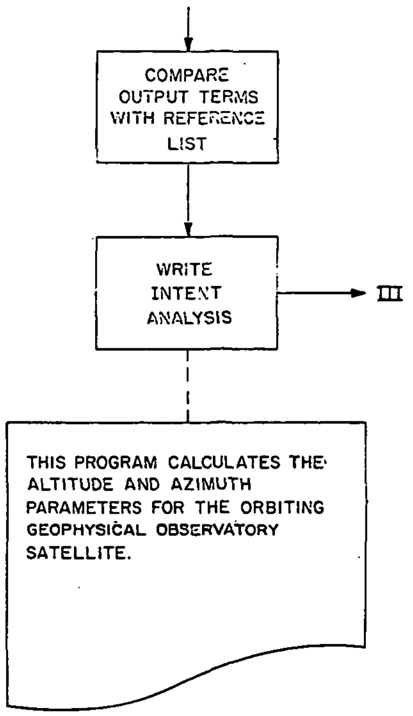

COMPARE OUTPUT TERMS WITH REFERENCE LIST
WRITE INTENT ANALYSIS
III
THIS PROGRAM CALCULATES THE ALTITUDE AND AZIMUTH PARAMETERS FOR THE ORBITING GEOPHYSICAL OBSERVATORY SATELLITE.

Figure 7.—Step 2 of an automated analyzer: Semantic intent analysis.

One other interesting complexity exists in this example, the presence of the same variable on both sides of an assignment. This should pose no problem in a straight assignment analysis; however, in a loop, if there were no prior reference to the variable outside, it would have to be treated as a generating point.

An automated system similiar to the one described in this paper would combine the following four previously defined functions: (1) a description of each output data set, (2) a description of the original intent of the program by analysis of the output terminology, (3) an analysis of each output-producing module, and (4) a description of all input data.

The system, after scanning the source deck of a program (fig. 6), would first process all output-related statements. From these it would produce a sample of each printed output in the program, replacing the name of the variable and its format for each output data item. For nonprinted output, the system would produce a description of the data set.

From the output statements, the system would also retrieve all significant label terms (fig. 7). It would compare these against a dictionary of terms tailored to a specific area and produce a complete description of the application of each program or segment. This part of the system could readily be merged with existing information retrieval systems, such as NASA's RECON system, which would serve as the limited-context dictionary necessary to analyze the output terminology.

This automated system would then begin an analysis of each segment of the program (fig. 8). Starting at each output data element, it would trace the pattern of calculations

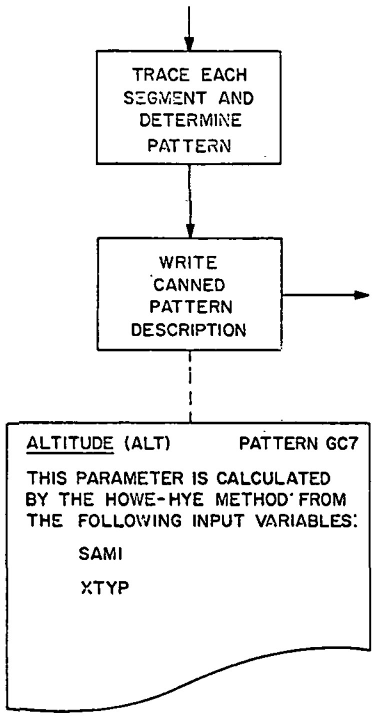

TRACE EACH SEGMENT AND DETERMINE PATTERN
WRITE CANNED PATTERN DESCRIPTION
IV
ALTITUDE (ALT) PATTERN GC7
THIS PARAMETER IS CALCULATED BY THE HOWE-HYE METHOD FROM THE FOLLOWING INPUT VARIABLES:
SAMI
XTYP

Figure 8.—Step 3 of an automated analyzer: Content analysis.

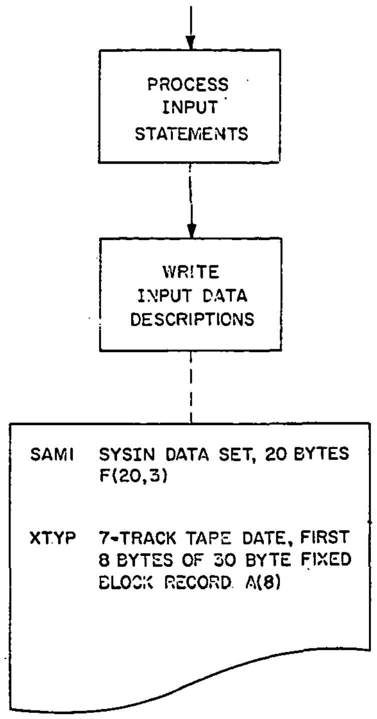

PROCESS INPUT STATEMENTS
WRITE INPUT DATA DESCRIPTIONS
SAMI SYSIN DATA SET, 20 BYTES F(20,3)
XTYP 7-TRACK TAPE DATE, FIRST 8 BYTES OF 30 BYTE FIXED BLOCK RECORD A(8)

Figure 9.—Step 4 of an automated analyzer: Input data description.

using list processing techniques and reduce each to its simplest form. It would then compare these to its memory of patterns and, upon finding a match, would print prewritten descriptions of each pattern. It would also indicate what variables were input or generating points for each pattern.

Finally, it would process the input statements (fig. 9), extracting such information as it could from them in a manner similar to the processing of nonprinted output data, and would print out descriptions of each of these input items.

A system such as the one described appears to be technologically feasible now, but, to the best of my knowledge, does not exist commercially at this time. Its development, if not already under way, should soon be undertaken.

DISCUSSION

MEMBER OF THE AUDIENCE: This type of approach will only work on certain types of programs, and I was wondering if you could characterize these a little better. For example, in a simulation program, it is going to be practically impossible to trace the origin of the statistical result back through the previous simulation process. A lot of programs come in several steps in which the output is really dependent on many previous calculations that do not show up in the output. Can you characterize the type of programs that you expect us to work on?

ARNOLD: I have to admit that our original thoughts were based on scientific and business programs and not simulation programs. However, I think that it is certainly feasible to apply this to almost any class with enough effort. Although it might be difficult, it is probably within the realm of possibility to apply it even to your case.

MEMBER OF THE AUDIENCE: If you can do this, then you should also perhaps set your sights a little higher. When you accomplish this, you will also be able to verify the correctness of the programs. If you can really trace the output back through the input, you should be able to verify while you are at it, too, I should think.

ARNOLD: That would be a very excellent adjunct to such a system.

MEMBER OF THE AUDIENCE: That is a very difficult process.

ARNOLD: Yes.

Page Intentionally Left Blank

N73-19248

COST ADVANTAGES OF AN INTEGRATED DOCUMENTATION APPROACH

William O. Felsman
Litton Systems, Inc.

Interactive use of on-line computing terminals is a method of increasing importance in developing or updating programs. Two major advantages are realizable: the turnaround time is decreased by at least one order of magnitude, and the programmer learns to treat the computer not only as a computational device but also as a combination blackboard and library. All additions and updatings are consequently performed directly on disc storage, and the completed program is thus available only in the raw form in which it was developed.

Program cleanup is required before formal documentation. TIDY and SWAP are two programs that assist this function. Figure 1 shows a subroutine that is representative of the kind of source code that can result when using on-line, interactive programming techniques, particularly if full advantage is made of the on-line system by assigning several programmers to related portions of the same task.

The TIDY program reassigns statement numbers in ascending order, with the base number and number increment being defined by the operator. It also generates a standardized, closed-up source language format. Figure 2 shows the RAW subprogram after being processed by TIDY.

```
      SUBROUTINE    SUB3( I)
       DIMENSION  IARRY(32 )
      IEMP1=I
      IEMP2=   1073741824
         IF(I)77, 10, 10
77    I= I +2147483647+  1
       IARRY( 1)= 1
       GO TO  11
 10   IARRY( 1)=0
  11 DO   16    J=1,31
     IF (I  - IEMP2)12,88,88
  12 IARRY(J+ 1)=0
     GO TO 90
88   IARRY(J+ 1)= 1
      I=I-IEMP2
 90  IEMP2=IEMP2/2
  16 CONTINUE
     I= IEMP1
     WRITE(6,800) (IARRY(J1) ,J1=1,32 )
 800 FORMAT(1X, 4(4I1, 1X, 4I1 ,2X ))
     RETURN
     END
```

Figure 1.–RAW subprogram.

Two further transformations are useful for improving the program legibility. First, the equal, plus, and minus operators can be set off by blanks and the variable names can be replaced by others of increased semantic content. Figure 3 shows the TIDY'ed program after being thus processed by SWAP. The format is more pleasing, and the variables now indicate the function they serve. For example, the name of the subroutine BISHOW shows some relation to its function of generating a bit-by-bit printout of the contents of the named variable, and so on.

PREDOCUMENTED SUBROUTINES

The use of predocumented subroutines in the development of a new

```
      SUBROUTINE SUB3(I)
      DIMENSION IARRY(32)
      IEMP1=I
      IEMP2=1073741824
      IF(I)10,12,12
   10 I=I+2147483647+1
      IARRY(1)=1
      GO TO 14
   12 IARRY(1)=0
   14 DO 22 J=1,31
      IF(I-IEMP2)16,18,18
   16 IARRY(J+1)=0
      GO TO 20
   18 IARRY(J+1)=1
      I=I-IEMP2
   20 IEMP2=IEMP2/2
   22 CONTINUE
      I=IEMP1
      WRITE(6,24)(IARRY(J1),J1=1,32)
   24 FORMAT(1X, 4(4I1, 1X, 4I1 ,2X ))
      RETURN
      END
```

Figure 2.–TIDY'ed program.

```
      SUBROUTINE BISHOW(NUMBER)
      DIMENSION MEMBIT(32)
      NUMSAV = NUMBER
      NEXT = 1073741824
      IF(NUMBER)10,12,12
   10 NUMBER = NUMBER + 2147483647 + 1
      MEMBIT(1) = 1
      GO TO 14
   12 MEMBIT(1) = 0
   14 DO 22 ICOUNT = 1,31
      IF(NUMBER - NEXT)16,18,18
   16 MEMBIT(ICOUNT + 1) = 0
      GO TO 20
   18 MEMBIT(ICOUNT + 1) = 1
      NUMBER = NUMBER - NEXT
   20 NEXT = NEXT/2
   22 CONTINUE
      NUMBER = NUMSAV
      WRITE(6,24)(MEMBIT(J1),J1 = 1,32)
   24 FORMAT(1X,4(4I1,1X,4I1,2X))
      RETURN
      END
```

Figure 3.–TIDY'ed and SWAP'ed program.

Table 1.–List of Common Subroutines

Compiler	Assembler	LOG2	SADL	SWAP	TIDY	Subroutine	Definition
X	X			X	X	PAGE	Page control
		X	X	X	X	INBUF	Source language read control
				X	X	OUTBUF	Source language standard output format
X	X	X	X	X	X	MOVER	Right adjust name
X	X	X				MOVEL	Left adjust name
	X			X		MVLEFT	Left adjust name, return spaces shifted
	X					MVRITE	Right adjust name, return spaces shifted
	X	X		X	X	ILSHFT	Shift name specified number of places to the left
	X	X	X	X	X	IRSHFT	Shift name specified number of places to the right
	X			X		IFXPT	Alpha to integer conversion
	X	X	X	X	X	DLIMIT	Delimit a source statement
	X	X	X	X	X	PACK3P	Terse number equivalence to variable name
	X	X	X	X	X	PACKP3	Reassemble names after delimiting
	X			X	X	UNPACQ	A4 format to A1 format
	X					ORDER	Indirect reference ordering of data

program is a well-known way to greatly reduce the attendant documentation effort at the same time that it reduces the program development time. Table 1 shows the extent to which predocumented subroutines were useful in the development of the operational programs discussed in this paper. The effect is similar to the use of a special higher order language, except that the more powerful operations are defined by subprograms rather than operators.

METAPROGRAM CONCEPT

Considerable additional advantage can be obtained if programs are written in a generic manner. For example, in an assembler program for an avionics computer, completion of the truncated operand addresses is normally a function of the attendant instruction. The alternatives might be to use the most significant bits of the instruction counter, to use a maximal length base address register, or to use a minimal base address register. If the usage for each instruction is written into the source code, then modification of the program to perform the assembly function for a second computer requires not only that the source code be redeveloped but also that the documentation be rewritten both at the program level and the user's manual level.

If, on the other hand, the basic program is written to provide for all of these potential choices and the branch chosen for a particular command is determined by a data set including the command mnemonic and a code defining the method of address completion, then this portion of the program needs no rewriting or redocumentation when the target computer is modified or a new target computer developed. Change of the data-set entry table is all that is required, together with the attendant minimal documentation.

This method of identifying significant parameters in a class of programs and then writing a generic program to accommodate these parameters in a defining data set is known as a "metaprogram" approach. That is, the data set is itself in effect a program, written in some very simple interpretive language and consequently has become known as the metaprogram.

Figures 4 and 5 show the metaprograms used for TIDY and SWAP.

The TIDY metaprogram is categorized by an identifying operator and several parameters. These define the start and finish locations within a FORTRAN statement of the series of places where statement numbers occur in that statement. The main program is thus

```
   DO    2   2
   GO    3   3    1
   GO    4  -4
 GOTO    2   2    1
 GOTO    3  -4
   IF   -1  -1    1
   IF   -5  -1
CYCLE    2   2
 READ    5   5
WRITE    5   5
```

Figure 4.–TIDY metaprogram.

```
GO=GO DELIMIT
TO=TO DELIMIT
CALL=CALL DELIMIT
REAL=REAL DELIMIT
INTEGER=INTEGER DELIMIT
SUBROUTINE=SUBROUTINE DELIMIT
DIMENSION=DIMENSION DELIMIT
DOUBLE=DOUBLE DELIMIT
PRECISION=PRECISION DELIMIT
DO=DO DELIMIT2
COMPLEX=COMPLEX DELIMIT
COMMON=COMMON DELIMIT
CYCLE=CYCLE DELIMIT
GOTO=DELIMITL GO DELIMIT TO DELIMIT DELIMITR
EQUIVALENCE=EQUIVALENCE DELIMIT
FUNCTION=FUNCTION DELIMIT
```

Figure 5.–SWAP metaprogram.

specifically directed to the locations where replacement is to occur. Additional statement types can be processed by adding to the metaprogram with no change in either the main program or the documentation associated with it.

The SWAP metaprogram is equally simple. The mnemonic to be replaced occurs to the left of the equal sign, and the replacing mnemonic sequence is to the right of the equal sign. Additional control parameters are included at the right. Thus, DELIMIT indicates a blank is to be inserted after each occurrence of a new variable in the output, and DELIMIT2 indicates that blanks are to be placed after both the replacing variable and the next succeeding variable. Additional control commands close up blocks, remove parentheses, and perform other functions.

This metaprogram is also open-ended, and additional statements in any of several higher order languages can be accommodated by appropriate descriptions in the metaprogram.

In fact, the data representing the desired mnemonic exchange is also treated as a simple continuation of the normal metaprogram.

```
STA1  CLA2  18001212  18001010
STA1        18001010
LDQ1  STA2  24001212  24001010
LDQ1        24001010
CLA2  ADD1  25012121  28001010
CLA2  SUB1  29012121  28001010
CLA1  STA2  28001212  28001010
CLA1        28001010
CLA2  MPY1  19012121  28001010
INC1  TRZ2  14001515  14001010
STQ1  STA2  16001212  16001010
STQ1        16001010
ADD1  STA2  26001212  26001010
ADD1        26001010
SUB1  STA2  27001212  27001010
SUB1        27001010
CLA2  ANA1  31012121  28001010
ANA1        31001010  31001010
CLA2  ORA1  30012121  28001010
ORA1        30001010  30001010
CLA2  DIV1  17012121
DIV1        17001010  17001010
CLA2  TRA1  00012323  00003010
TRA1        00003010
TNZ1  CLA2  01003232  01003010
TNZ1        01003010
TZA1  CLA2  05003232  05003010
TZA1        05003010
TPA1  CLA2  06003232  06003010
TPA1        06003010
```

Figure 6.–Typical assembler metaprogram (partial).

Figure 6 shows a portion of a metaprogram for an assembler. It consists of a defining mnemonic or mnemonic pair, plus a series of numerals that define the transfers within the main program which control the development of the assembled code for that instruction mnemonic. The assembler program is very nearly invariant for a wide class of computers, consequently documentation of the main program can here, too, be unchanged with new applications. However, the interpretation of the metaprogram by the programmer requires a computer assist to documentation.

Figure 7 shows the metaprogram as processed for semantic clarity, and figure 8 shows a portion of the documentation that relates the metaprogram controls to the operation of the computer arithmetic unit. The matrix documentation shown in figure 8 has been developed to reduce the difficulty in the handling, updating, and correction of bulk data. The matrix documentation program accepts data as a sequential input stream and formats it in both vertical and horizontal directions, with text hyphenation where applicable. Program control cards direct the number and width of the columns. Thus, each entry in any column is separately modifiable, and the program adjusts the full updated data input to maintain the format.

INTEGRATED DOCUMENTATION SAVINGS

Two examples of the cost of documentation are shown in table 2.

In each case, the compiler and assembler documentation for a given computer required full page counts of 140 and 95, respectively. However, for all successive applications to

		NON ZERO SECONDARY								ZERO SECONDARY							
					DIRECT		INDIRECT						DIRECT		INDIRECT		
PRI	SEC	NUMB	TYPE	M	PRI	SEC	PRI	SEC	SPR	NUMB	TYPE	M	PRI	SEC	PRI	SEC	SPR
STA1	CLA2	18	0	0	1	2	1	2	0	18	0	0	1	0	1	0	0
STA1		18	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
LDQ1	STA2	24	0	0	1	2	1	2	0	24	0	0	1	0	1	0	0
LDQ1		24	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
CLA2	ADD1	25	0	1	2	1	2	1	0	28	0	0	1	0	1	0	0
CLA2	SUB1	29	0	1	2	1	2	1	0	28	0	0	1	0	1	0	0
CLA1	STA2	28	0	0	1	2	1	2	0	28	0	0	1	0	1	0	0
CLA1		28	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
CLA2	MPY1	19	0	1	2	1	2	1	0	28	0	0	1	0	1	0	0
INC1	TRZ2	14	0	0	1	5	1	5	0	14	0	0	1	0	1	0	0
STQ1	STA2	16	0	0	1	2	1	2	0	16	0	0	1	0	1	0	0
STQ1		16	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
ADD1	STA2	26	0	0	1	2	1	2	0	26	0	0	1	0	1	0	0
ADD1		26	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
SUB1	STA2	27	0	0	1	2	1	2	0	27	0	0	1	0	1	0	0
SUB1		27	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
CLA2	ANA1	31	0	1	2	1	2	1	0	28	0	0	1	0	1	0	0
ANA1		31	0	0	1	0	1	0	0	31	0	0	1	0	1	0	0
CLA2	ORA1	30	0	1	2	1	2	1	0	28	0	0	1	0	1	0	0
ORA1		30	0	0	1	0	1	0	0	30	0	0	1	0	1	0	0
CLA2	DIV1	17	0	1	2	1	2	1	0	0	0	0	0	0	0	0	0
DIV1		17	0	0	1	0	1	0	0	17	0	0	1	0	1	0	0
CLA2	TRA1	0	0	1	2	3	2	3	0	0	0	0	3	0	1	0	0
TRA1		0	0	0	3	0	1	0	0	0	0	0	0	0	0	0	0
TNZ1	CLA2	1	0	0	3	2	3	2	0	1	0	0	3	0	1	0	0
TNZ1		1	0	0	3	0	1	0	0	0	0	0	0	0	0	0	0
TZA1	CLA2	5	0	0	3	2	3	2	0	5	0	0	3	0	1	0	0
TZA1		5	0	0	3	0	1	0	0	0	0	0	0	0	0	0	0
TPA1	CLA2	6	0	0	3	2	3	2	0	6	0	0	3	0	1	0	0
TPA1		6	0	0	3	0	1	0	0	0	0	0	0	0	0	0	0

Figure 7.–Processed assembler metaprogram (partial).

CODE	MNEMONIC	INSTRUCTION CONDITIONS M1 AND M2 FIELD USED NO INDIRECT ADDRESS	INSTRUCTION CONDITIONS M1 FIELD ONLY NO INDIRECT ADDRESS	INSTRUCTION CONDITIONS M1 AND M2 FIELD USED INDIRECT ADDRESSING	INSTRUCTION CONDITIONS M1 FIELD ONLY INDIRECT ADDRESSING
CC01	TRA M1 CLA M2	(M2+BARL) --> A PACK(PC,BAR(7))-->DED M1+PCS --> PC Q --> Q BARL --> BARL BARS --> BARS	 PACK(PC,BAR(7))-->DED M1+PCS --> PC Q --> Q BARL --> BARL BARS --> BARS	((M2+BARL)) --> A PACK(PC,BAR(7))-->DED M1+PCS --> PC Q --> Q BARL --> BARL BARS --> BARS	 PACK(PC,BAR(7))-->DED (M1+BARS) --> PC Q --> Q. BARL --> BARL BARS --> BARS
CC02	TNZ M1 CLA M2	M1+PCS -->PC (A NON 0) (M2+BARL) --> A PACK(PC,BAR(7))-->DED Q --> Q BARL --> BARL BARS --> BARS	M1+PCS -->PC (A NON 0) PACK(PC,BAR(7))-->DED Q --> Q BARL --> BARL BARS --> BARS	M1+PCS-->PC (A NON 0) ((M2+BARL)) --> A PACK(PC,BAR(7))-->DED Q --> Q BARL --> BARL BARS --> BARS	(M1+BARS)-->PC(A NO 0) PACK(PC,BAR(7))-->DED Q --> Q BARL --> BARL BARS --> BARS
CC03	LBR M2 TRA M1	M2 --> BAR (7 BITS) M1+PCS(NEW) --> PC Q --> Q BARL(NEW) --> BARL BARS(NEW) --> BARS PC+1 --> PC	 (M1+PCS(OLD)) --> PC Q --> Q BARL(OLD) --> BARL BARS(OLD) --> BARS PC+1 --> PC	(M2+BARL) --> BAR(7) M1+PCS(NEW) --> PC Q --> Q BARL(NEW) --> BARL BARS(NEW) --> BARS	 ((M1+BARS(OLD)))-->PC Q --> Q BARL(OLD) --> BARL BARS(OLD) --> BARS
CC04	SPC M1 CLA M2	(DED)--> (M1+BARS) (M2+BARL) --> A Q --> Q BARL --> BARL BARS --> BARS PC+1 --> PC	(DED)--> (M1+BARS) Q --> Q BARL --> BARL BARS --> BARS PC+1 --> PC	(DED)--> (M1+BARS) ((M2+BARL)) --> A Q --> Q BARL --> BARL BARS --> BARS PC+1 --> PC	(DED)--> ((M1+BARS)) Q --> Q BARL --> BARL BARS --> BARS PC+1 --> PC
CC05	STA M2 TRA M1	(A) --> (M2+BARL) PACK(PC,BAR(7))-->DED M1+PCS --> PC Q --> Q BARL --> BARL BARS --> BARS	 PACK(PC,BAR(7))-->DED M1+PCS --> PC Q --> Q BARL --> BARL BARS --> BARS	(A) --> ((M2+BARL)) PACK(PC,BAR(7))-->DED M1+PCS --> PC Q --> Q BARL --> BARL BARS --> BARS	 PACK(PC,BAR(7))-->DED (M1+BARS) --> PC Q --> Q BARL --> BARL BARS --> BARS
CC06	TZA M1 CLA M2	(M2+BARL) --> A PACK(PC,BAR(7))-->DED M1+PCS-->PC (A=0) Q --> Q BARL --> BARL BARS --> BARS	 PACK(PC,BAR(7))-->DED M1+PCS-->PC (A=0) Q --> Q BARL --> BARL BARS --> BARS	((M2+BARL)) --> A PACK(PC,BAR(7))-->DED M1+PCS -->PC (A=0) Q --> Q BARL --> BARL BARS --> BARS	 PACK(PC,BAR(7))-->DED (M1+BARS)-->PC (A=0) Q --> Q BARL --> BARL BARS --> BARS
CC07	TPA M1 CLA M2	(M2+BARL) --> A PACK(PC,BAR(7))-->DED M1+PCS -->PC (A=+) Q --> Q BARL --> BARL BARS --> BARS	 PACK(PC,BAR(7))-->DED M1+PCS -->PC (A=+) Q --> Q BARL --> BARL BARS --> BARS	((M2+BARL)) --> A PACK(PC,BAR(7))-->DED M1+PCS -->PC (A=+) Q --> Q BARL --> BARL BARS --> BARS	 PACK(PC,BAR(7))-->DED (M1+BARS)-->PC (A=+) Q --> Q BARL --> BARL BARS --> BARS

Figure 8.–Matrix documentation of assembler metaprogram (partial).

Table 2.–Documentation Page Count

Documentation stage	Compiler	Assembler
Theory of operation	70	40
Flowcharts	30	30
Description of charts	30	20
Metaprogram description	10	5
Total	140	95

Table 3.–Program Organization Statistics, Source Code

Name	Basic program length (lines)	Total program length (lines)	Development time (man-months)	Basic program[a]	Separable program-peculiar subroutines[a]	Data set[a]	Predocumented subroutines[a]
TIDY (statement number reorder)	229	686	0.75	0.334	0.052	0.015	0.599
SADL (reliability model)	475	816	1.5	.590	.025	.007	.378
LOG2 (logic simulator)	515	857	1.5	.602	.000	.003	.395
SWAP (mnemonic exchange)	210	637	.5	.330	.000	.024	.646
Memory allocator	587	887	3	.662	.025	.016	.298
Assembler	829	2193	5.5	.378	.077	.063	.282
Compiler	1216	1690	24	.720	.063	.200	.017

[a]Proportions of total programs.

differing computers, the compiler required only a new description of the metaprogram, and the assembler, which was less completely organized into a generic program, required a new metaprogram description plus about 30 percent of the other page counts.

Some indication of the savings to be gained by the use of predocumented subroutine and metaprogram data-set techniques can be seen in table 3, which gives the relative usage of these methods for a variety of programs. Documentation savings run typically from 30 to 60 percent on the initial program development. However, the use of subprograms has a major effect upon the program cost itself. Table 3 tabulates the number of lines of source code for the programs investigated together with the development time for these programs. These programs were developed by a group of five people, of consistent skill level, working both individually and as members of small teams. A plot of these data is shown in figure 9.

The most significant aspect of the graph is that it shows an exponential growth of development time with the length of the basic program. The basic program consists of the main program plus such program-peculiar subroutines as are conceptually involved with the main program in a highly complex manner. Thus, implementation of an integrated

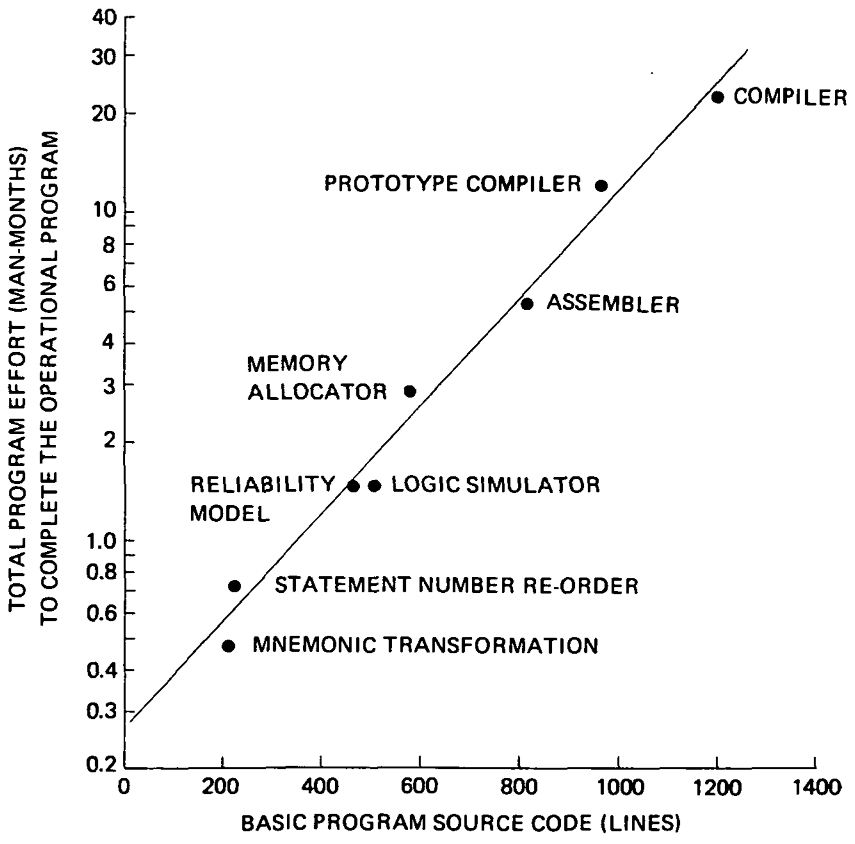


Figure 9.–Program development time.

documentation approach, which suggests the use of small, preferably predocumented, subprogram elements is not inconsistent with the program design cycle, which also shows greater efficiency in the time of program development when appropriate organization permits the size of the basic program to be reduced.

MODIFICATION

Figure 10 shows the advantages resulting from the use of the integrated documentation approach when it was necessary to modify an existing assembler to accommodate a new computer. Two basic programs were available for modification, one with both subroutines and a metaprogram, the other being predominantly large program elements. Eighty-five percent of the source code from the integrated approach was applicable, whereas only 40 percent of the unitized code could be reused.

Figure 10 also shows the costs associated with the two modifications. The unitized code changeover was estimated at 15 man-months. The actual cost to update the program using an integrated documentation approach was 2 man-months.

Further cost savings are shown in table 4. Here, the effect of the metaprogram is detailed. The cost of modification from one target computer to another is compared to the

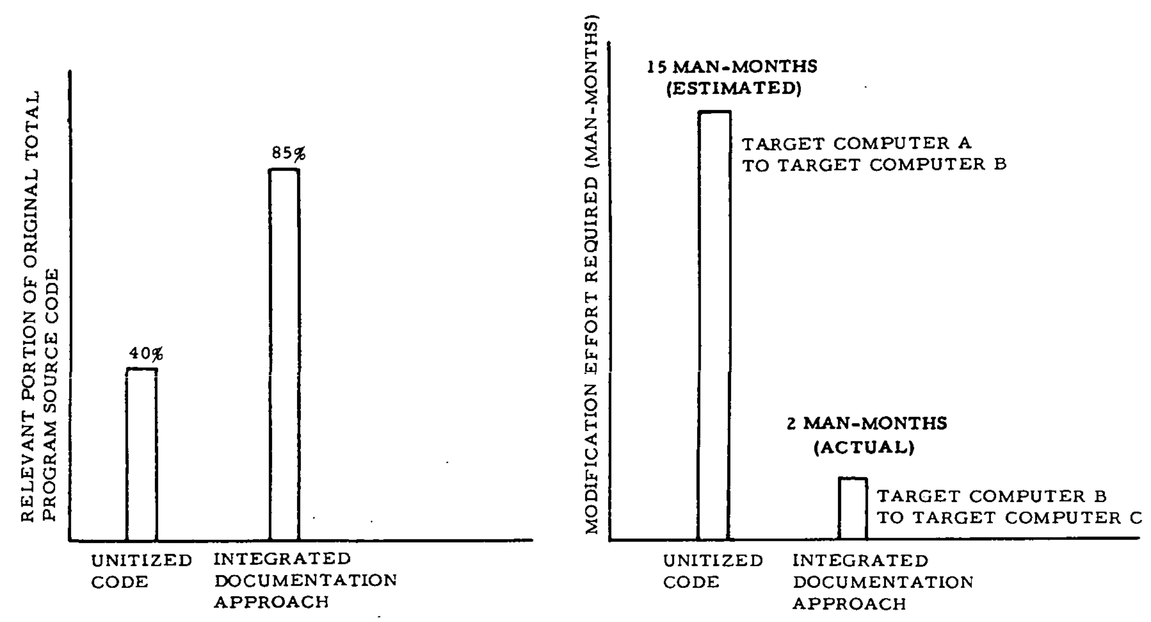


Figure 10.—Program modification comparison (assembler program).

Table 4.—Integrated Documentation Effects on Program Modification

Program type	Data set proportion in source code	Rate of cost of modification to new program design
Compiler	0.20	0.1
Assembler	.06	.3

cost of a new program design for the new target computer. Examples are given for a compiler and an assembler.

The compiler, which makes extensive use of a metaprogram, can be modified to accommodate a new target computer at one-tenth the cost of writing a new compiler program. Modifications are almost entirely to the metaprogram. The assembler, which uses a less well-developed metaprogram but which does make extensive use of common subroutines, can be modified to accommodate a new target computer at about one-third the cost of developing a new assembler program.

RUNNING TIME

Run time of production programs developed using the integrated documentation approach would appear at first consideration to be somewhat higher than the run times of similar programs written in unitized code. This is because the use of prepackaged subroutines implies a close, but not exact, fit between the requirement and the package and because of the extra time required to execute the branch statements implied by metaprogram techniques.

However, the integrated documentation approach tends to segment a program into functionally consistent elements with minimal communication required between them and consequently parallels good programming practice. The result is that the total program size is somewhat reduced, values from 10 to 30 percent being typical. Smaller programs, using less core, generally are charged a lesser main frame usage rate. This lesser rate compensates for the longer execution time.

CONCLUSION

An integrated documentation approach using predocumented subroutines and metaprogram techniques is a very efficient means of not only generating the relevant documentation but also of reducing program development costs.

DISCUSSION

MEMBER OF THE AUDIENCE: Do you use this approach on all your programs? In other words, do you think that all programs are divisible into small metaprograms?

FELSMAN: Yes, they are divisible, but we do not always do it because occasionally you run into someone who wants something in a hurry. Then we simply write in a standard fashion as rapidly as we can. It is a one-of-a-kind thing, and we do not worry about documentation. But for big problems like compilers, assemblers, and memory allocators, we go through the process and do indeed break it out, use our regular subroutines, and always write data-set-wise or metaprogram-wise. It is much more convenient.

MEMBER OF THE AUDIENCE: I think your presentation answered the question raised by Gridley yesterday about whether we should put emphasis on subroutines or smaller parts. At that time we did not really respond to his question on the panel, and I think we should have because it is an asset not only in developing programs but in distributing programs to other people to use. If you are going to use an entire program without modifying it, fine. But when you develop programs, I think your point is valid that it takes less time to develop a new program from well-documented subroutines or metaprograms than it does to modify an existing program to make it work on a computer other than the one for which the program was written.

MEMBER OF THE AUDIENCE: You implied that this was for a given set of target computers.

FELSMAN: Yes. In this case they happen to be all airborne computers.

MEMBER OF THE AUDIENCE: Have you considered it for a general-purpose type of computer in a general-purpose environment?

FELSMAN: As far as I can tell, we have investigated other military computers like the AN/UYK-7, which is a very powerful floating-point machine very similar to some of our commercial machines, and we wrote a metaprogram for that using this compiler. I think the answer to your question is yes, although not unequivocally.

MEMBER OF THE AUDIENCE: Presume a manufacturer has provided this to us, but our problem is related to application programs. Can this be applied there?

FELSMAN: You have a more difficult task because you have to find the common parameters from application to application. If you can find any, you can do it. If they are not immediately evident, maybe you cannot do it.

N72-19219

AUTOMATIC PROGRAM ANNOTATION (AUTONOTE)

Michael D. Neely
and
Judy W. Tyson
ARIES Corp.

Computer program documentation, the "maps" of the computing industry, is a very neglected area of this industry. Countless hours are spent trying to find out what has been done in the past, and time and money are wasted duplicating past efforts.

However, the poor quality of these "maps" is only a secondary effect of this neglect of the field. The primary effect is the scarcity of tools that can be used in the documentation process. A determination of what types of tools are needed still must be done. This paper will attempt a preliminary identification of these tools.

Before those tools can be identified, though, the uses of the documentation to be produced must be determined. The most important uses are program maintenance, which includes enhancements and error detection and correction, and program development. Efforts in these areas to develop tools that produce good documentation will be more than repaid.

This paper will try to identify some areas of program documentation that should be automated and then will focus on a particular area and explore the possibilities of automation. In general, this discussion is directed at the assembly language program, but in some areas the remarks are germane to metalanguages. The emphasis will be on the tools that are needed, not on the means of providing those tools.

BASIC REQUIREMENTS

Program documentation usually consists of the program specifications, flowcharts, program listing, and operating instructions. Because the specifications are written before the program is and because means of automatically flowcharting programs are already available, the program listing and operating instructions will be the topics discussed here.

The program listing contains coding, the machine language instructions generated by the coding, and comments relating to the coding. When properly interpreted, the coding supplies the most accurate "map" of the program. Interpreting the coding is one of the areas in which automation can improve the use of the listing. The operating instructions provide the information necessary to use the program. They specify the interface between the program, the operator, and the peripheral devices. Some of the operating instructions can be produced as a byproduct of interpreting the coding.

Most of the information needed by the maintenance programmer can be produced by an analysis of the coding. This includes a set of consistent comments relating to the coding, error identification, executive function identification, analysis of arithmetic operations, and various cross-reference tables. Much of this same information is necessary in the development of a program library but, in addition, it might be desirable to produce other types of information, more general in content. These can be produced by the use of control cards to identify the information. We will consider each of these areas to determine what effect automation can have on the documentation process.

Comments

Program listings normally include comments provided by the programmer; in addition to these comments, a means of automatically producing comments from the coding would be helpful. These comments could be on two different levels: An analysis of the coding or an analysis of the logic defined by the coding. This area will be explored in more detail in a later section of this paper.

Error Detection

The objective of the error detection process would be to identify those errors that are readily apparent from an analysis of the coding without going into a detailed analysis of the logic of the program. It would be impossible, for example, to determine from the coding if a program meets the program specifications. It would be possible, however, to spot other errors that are related to the mechanics of coding.

Errors that could be readily detected include instructions using invalid operators, invalid operands, undefined program labels, or doubly defined program labels. For bank-oriented computers it would be useful to identify areas where code spills over the end of a bank or where an area of core is overlaid. Another possibility in this area would be the flagging of instructions referencing items located in a different bank.

A different type of error would involve the use of computer registers. It would be possible to flag coding in which the contents of a register are destroyed. In this case, the register is loaded with one value, then loaded with another value before the first value has been used. In double-precision operations it would be possible to flag instructions that reference improperly aligned items.

Errors should be flagged whenever they are encountered in the coding; in addition there should be an error summary at the end of the listing.

Operator Interface

Executive functions, mainly input- and output-related operations, are an important segment of any program and therefore play an important role in the program documentation. It is necessary to specify which devices are used, the manner in which they are used, and what operator interface is required for those devices. All these can be provided automatically; in addition, AUTONOTE could associate calls to a particular device with the coding, thus producing a cross-reference table of input/output (I/O) calls by device. It might also be possible to associate buffer areas with the devices using those areas.

Another interesting possibility is the identification of all input and output operations on a particular data set. However, this would be more easily implemented with metalanguages than with assembly languages.

In the area of console communications it would be desirable to associate operator messages with the coding that produces the messages. When a response is required from the operator, all valid responses should be listed and default responses should be specified.

Arithmetic Operations

For arithmetic operations, the documentation should specify the limits imposed on an operation by the machine word size, mode of operation (single or double precision), or constants that are used. Iterative operations should be identified, and the limits on the number of iterations should be specified. These functions can be provided from an analysis of the coding. In addition, it may be possible to analyze algorithms to produce the formulas defined in the coding. In metalanguages it would be possible to specify the accuracy that would be obtained in an arithmetic process.

Cross-References

Program listings normally have only one cross-reference table, an alphabetical list of program labels with the instruction numbers in which the labels are referenced. Several other types of cross-reference tables would be useful and could be produced easily during an analysis of the coding. In addition to the basic alphabetical cross-reference of all items, there should be separate cross-reference tables for data constants, address constants, buffer areas, subroutines, I/O calls, and labeled instructions. The tables of constants should identify any duplicated items, and all tables should identify unreferenced items. In addition, there should be a separate cross-reference table of undefined items. These various tables would aid the programmer in debugging his program originally as well as in maintaining the completed program.

When origin instructions to the assembler program are provided, AUTONOTE would produce a basic core map showing the area used, the location of all origin statements, and the location of any overlaid areas of core.

These are all basic items that should be provided in the documentation to aid the maintenance programmer. However, in addition to these items, many organizations require that documentation be in a specified format. The information required for this documentation is usually, or at least should be, included in the program listing in the form of comments. This information includes program name, acronym, organization name, programmer, assembly date, equipment configuration, source language, core requirement, execution time, and program abstract. With the use of control cards, this information can be extracted from the program to produce the documentation in the specified format. However, this capability should be included only as an option, and the use of control cards should not be necessary to use the other features of AUTONOTE.

AUTOMATIC COMMENTS

The possibilities of automation in the area of program comments should be explored further. This area is particularly susceptible to automation because the current method, relying on programmer-supplied comments, has several inherent disadvantages: Not all programmers comment programs in the same detail; the comments may be meaningful only to the original programmer; and often program coding is changed, but the comments remain the same. In addition, in metalanguages the programmer may not be familiar with the assembly language that is generated and may not be able to determine from the coding what is taking place.

To automatically comment a program, first, a set of comments would be associated with the instruction set, as shown in figure 1. Instructions that cause an alteration in the sequential execution of code under certain circumstances would require more than one comment. These comments would then be used to explain the mechanics of the operations. An illustration of this technique is shown in figure 2. This process requires a limited ability to look ahead in the coding to determine what is taking place, but it does not require a detailed analysis of the logic defined by the coding. The automatic comments are provided in addition to the programmer's comments, not in place of those comments.

It would also be possible to comment a program by analyzing the logic, roughly the procedure used by flowcharting programs, but that would require a greater effort in looking forward and backward in a program and would be a duplication of the flowcharting process. Because the program listing is considered a complement to the flowchart, production of comments keyed to the coding should be an area of concentration.

Subroutines are an important element in any program, and the comments relating to the subroutines are important for an understanding of the program. Good documentation of subroutines is not only helpful to the maintenance programmer but also important in the development of a subroutine library. Well-documented subroutines can prevent needless duplication of effort in the development of new programs. Figure 3 shows an illustration of a relatively simple subroutine with the programmer's comments. Figure 4 shows the same subroutine with the addition of automated comments. These comments not only describe the processing within the subroutine but also tell which program registers are loaded prior to entry, where the subroutine is called from, which items are for internal use, which external items are used, and which registers are modified at the exit. Much of this information can also be provided in a cross-reference of subroutines at the end of the program listing.

INSTR.	AUTOMATIC COMMENT
CRA	CLEAR A
STA	STORE A IN XXXX
LDA	LOAD A FROM XXXX
ADD	ADD A TO XXXX
SUB	SUBTRACT A FROM XXXX
ALS	SHIFT A LEFT X BITS
LLL	SHIFT B TO A LEFT X BITS
CAS	COMPARE A TO XXXX GREATER EQUAL LESS THAN
SMI	BIT 1 OF A=1 NO YES
SR1	SENSE SWITCH 1 SET YES NO
ENB	ENABLE INTERRUPTS

Figure 1.–Instruction set and associated comments.

```
STAK    LDA     *BUFA                           *LOAD A FROM INPUT BUFFER

        SNZ                                     *A=0

        JMP     $+2                             *YES-GO TO THIS LOCATION+2

        STA     *BUFB                           *NO-STORE A IN OUTPUT BUFFER

        IRS     BUFA      FILL OUTPUT BUFFER    *INCREMENT FWA INPUT BUFFER

        IRS     BUFB                            *INCREMENT FWA OUTPUT BUFFER

        IRS     NOWD                            *WORD COUNT=0

        JMP     STAK                            *NO-GO TO STAK

        JMP     EXIT                            *YES-GO TO EXIT


*

BUFA    DAC     INBF      FWA INPUT BUFFER

INBF    RES     10        INPUT BUFFER

BUFB    DAC     OTBF      FWA OUTPUT BUFFER

OTBF    RES     10        OUTPUT BUFFER

NOWD    DATA    -10       WORD COUNT
```

Figure 2.–Operation comments.

This type of documentation provides adequate information for maintenance programmers and the information needed to develop a program or subroutine library.

As for the format of the documentation, AUTONOTE would begin each listing with the set of instructions and their associated comments (fig. 1). Then would come the program with the automatic comments. These comments would not replace the programmer's comments but would be in a separate column. Thus, for each instruction, there could be two sets of comments. As an option, the programmer could use control cards to suppress the listing of automatic comments in sections of the program. Errors would be marked at the point of origin, and there would also be an error summary at the end of the listing with a separate list of error codes and their meaning. After that would come the cross-reference table of all items and the individual cross-references. Figures 5 and 6 show examples of cross-references for data constants and address constants. In addition to the standard list of label, location, and reference points, this list contains the value of the item and a list of

```
*
* CHECKSUM ROUTINE
* X=FWA
* A=LENGTH
*
CCCK     DAC     0
         TCA
         STA     CLGH
         CRA
         STA     CHEX
         LDA     *0
         ADD     CHEX      COMPUTE CHECKSUM
         STA     CHEX
         IRS     0
         IRS     CLGH      DONE
         JMP     $-5       NO
         JMP     *CCCK     YES
*
CLGH     DATA    0         CHECKSUM COUNTER
CHEX     DATA    0         CHECKSUM
*
```

Figure 3.—Programmer's subroutine comments.

duplicate items. For bank-oriented computers there would be a core map. If control cards are used to request lists in a specified format, these would follow the cross-reference tables.

ADDITIONAL CAPABILITIES

There are many possibilities for expanding a system such as AUTONOTE. For example, with the use of a cathode ray tube it would be possible to "page" through a program. The sequential flow of the program could be interrupted to look at subroutines, with the flow resuming at the end of the subroutine. It would also be possible to modify programmer comments at the on-line terminal or edit those comments to produce a program or subroutine abstract. Subroutines could be selected for a library in this manner, and the library could then be queried from the terminal.

Of course, a system like this presents many challenges as well as opportunities. There are several technical problems that would have to be resolved. For example, in assembly languages it is often difficult to identify I/O devices or to define record layouts. Indexing and indirect addressing would also present problems for the analysis.

The assembly language program has been the topic of this presentation; other types of programs present different problems and possibilities. In metalanguages it is easier to define record layouts and identify I/O devices, and it might be possible to identify and define blocks of logic, but detailed comments might be redundant, as most metalanguages are at least partially self-documenting. Another possibility would be to use the intermediate output of compilers, the assembly language program, as the input to AUTONOTE.

Means of reducing the cost of software must be developed if the software industry is to continue expanding as it has in the past. AUTONOTE may represent a step in the right direction. It will provide reliable, consistent documentation of programs, something that has been lacking in the past. It would be a useful tool for the maintenance programmer, it would

```
*                                                     *LOADED PRIOR TO ENTRY:
*                                                     *A,X
*                                                     *ENTERED FROM CARD NOS.
*                                                     *247,654
* CHECKSUM ROUTINE
*   X=FWA
*   A=LENGTH
*
CCCK      DAC       0                                 *RETURN ADDR CALLING PROG
          TCA                                         *TWO'S COMPLEMENT A
          STA       CLGH                              *STORE A IN CHECKSUM COUNTER
          CRA                                         *CLEAR A
          STA       CHEX                              *STORE A IN CHECKSUM
          LDA       *0                                *LOAD A INDIRECT FROM X
          ADD       CHEX      COMPUTE CHECKSUM        *ADD A TO CHECKSUM
          STA       CHEX                              *STORE A IN CHECKSUM
          IRS       0                                 *INCREMENT X
          IRS       CLGH      DONE                    *CHECKSUM COUNTER=0
          JMP       $-5       NO                      *NO-GO TO THIS LOCATION-5
          JMP       *CCCK     YES                     *YES-GO TO RETURN ADDR CALLING PROG
*
CLGH      DATA      0         CHECKSUM COUNTER
CHEX      DATA      0         CHECKSUM
*
*                                                     *INTERNAL ITEMS:
*                                                     *CLGH
*                                                     *EXTERNAL ITEMS:
*                                                     *CHEX
*                                                     *MODIFIED AT EXIT:
*                                                     *A,X
*                                                     *A=CHEX
```

Figure 4.—Automated subroutine comments.

aid in the development of program libraries, and as a bonus it would be useful as a debugging tool during the original development of a program. This is the type of tool that is needed to begin the war on soaring software costs.

```
* DATA CONSTANTS

A1      511     000001     497     618     627

A2      512     000002     480

A3      513     000003     502

A4      514     000004     312     510

A5      515     000005     452

A6      516     000006     379     580     603

A7      517     000007     411

A8      518     000010     701     718

.
.
.
.
.
.
.

AHIH    103     177777     123     347

ALOW    102     000001     121     214     390

.
.
.
.
.
.

*DUPLICATES:

*AL    ALOW                                        VALUE: 000001
```

Figure 5.—Cross-references for data constants.

```
* ADDRESS CONSTANTS:

AFIL      202      AFL1      218      327      339      409

BFIL      311      BFL1      114      256      423

CFIL      119      CFL1      489      522      679

DFIL      450      DFL1      560

.

.

.

.

.

.

.

.

.

.

ZFIL      332      AFL1      510      736      759      840

.

.

.

.

.

*DUPLICATES:

* AFIL   ZFIL                                          VALUE: AFL1
```

Figure 6.–Cross-references for address constants.

DISCUSSION

MEMBER OF THE AUDIENCE: I believe I disagree with a lot of what you said. I think that translating what the machine is doing into another statement does not help at all. This is what our documentation assembly level has done for so long. What we really need is information that attempts to describe what the machine is doing in relation to a definition of specifications of that job and how it relates to that part of the specifications. In other words, what you really need to know most of the time is: What was to be achieved?

NEELY: I agree. This is only going to tell you what he did achieve. I think debugging the program originally would be the main use of it. No doubt this is not a panacea for the industry or for the documentation process.

MEMBER OF THE AUDIENCE: The purpose of a comment is to describe the function, and I think that if you can get a well-commented program you are at least halfway to good documentation of that program. If you discourage the people or provide something that gives you essentially a description of the microcode, the fact that you are loading and storing really gives you very little. It is important not to have programs that are uncommented because one comment should describe maybe five or six instructions and give a functional description. If you encourage people not to do that, you are really going in the wrong direction.

NEELY: I agree with you that we should not replace the programmer's comments by any means, but this would be used in conjunction with his comments.

PANEL DISCUSSION

MEMBER OF THE AUDIENCE: I might start off with a question that Felsman and Kalar could discuss since they are actively supplying programs. What would you do in the case of a program giving the wrong answer? What is the responsibility of the distribution center in a case like this?

PANEL MEMBER: I think I can answer it because to a large extent inside Litton we do act as a semidistribution center. When something is wrong, the first thing we do is find out whether the program was run correctly. If it seems to have been, we find out what the problem is as soon as possible because we do not want to have the reputation that we do not know what we are doing.

MEMBER OF THE AUDIENCE: Do you have any procedure for redistributing new copies?

PANEL MEMBER: No official way, but we make sure that everybody who got the original also gets the followup.

PANEL MEMBER: At COSMIC, if we get a trouble report about a program, the programming manager, who is quite capable, tries to solve the problem. If he cannot do it, we contact the originator if possible. If we cannot get in touch with the originator, we try to find somebody at the center where the program came from who is familiar with the program and can help us.

MEMBER OF THE AUDIENCE: Earlier this morning Field said that this would be the first time he had heard a contractor wanting to tighten up the request for proposal (RFP). When it becomes profitable to document well, then I think you will not have to have a symposium to find out how to document automatically. I have never seen one of your RFP's, but I have seen RFP's that in fact required no delivery of documentation. I am sure that the customer wanted the documentation and thought he had asked for it. In fact, I have seen RFP's that required no delivery of the program. I have no objection at all to bidding on a project if it requires no delivery of the program.

If the users to whom the documentation is most profitable will pay a little more attention to making it profitable to the contractor and the program production people, I think there would be less of a documentation problem. When something has been actually implemented, it is because it is profitable and cheaper for a contractor to do the job right once. For some specific programs, it is not cheaper to document them, it is cheaper to write them and turn them over if you can. Does the panel have any comments on this?

PANEL MEMBER: I think we are trying to reach a point in the industry where one of the requirements is documentation with every RFP. You should not consider underbidding your opponents by leaving it out. This should be part of the requirement for everything, and its cost should be accepted just as programming costs are.

MEMBER OF THE AUDIENCE: I would think a responsible contractor might propose something that has been left out in the RFP as a separate item. Certainly you do not want to include it in your price and not point it out because that would hurt your competitive position, but I think it would be worthwhile either to check with the contracting officer about the minimum required or to include this little extra in your proposal.

MEMBER OF THE AUDIENCE: It is difficult to define standards, but I think they have to be defined in the RFP. When they are, there will be more automatic documentation.

PANEL MEMBER: I would agree. It is primarily the Government's responsibility. There is a considerable amount of programming going on at Goddard Space Flight Center, and stricter RFP's, as well as the usual policies and procedures and demand for documentation of a higher quality, could help in emphasizing documentation.

MEMBER OF THE AUDIENCE: Both the scope of the documentation and the depth of the documentation have to be balanced against cost, time, and effort. There is a wide range of documentation, and it is up to the customer to decide what degree of documentation is needed for his purposes.

PANEL MEMBER: Part of the problem is the fact that management in general is not aware of the cost of not documenting. Not documenting can be a very real problem, but there is a distinction between documenting for someone who is unfamiliar with the techniques that you have been using and documenting for someone who is very familiar with the techniques you have been using. In-house documentation tends to be sparse compared with what we provide the customer because our employees know our business. They do not need quite as much information to be able to pick up where someone else left off. But with someone else, it requires a good deal of additional information to get the point across so that the information is not lost. It is that extra part that the contractor has to keep paying for which we do not normally do.

PANEL MEMBER: It seems that there are two kinds of documentation. One kind makes the program usable to other people, and it seems that a number of excellent ways of automating methods for this kind of information have been proposed. The other kind is the kind that a user is going to need to modify these programs. Are there any comments on this second kind?

MEMBER OF THE AUDIENCE: Talking about RFP's and requirements, I think it is very difficult for the person who is writing an RFP to know exactly what he is going to get in the end because he cannot predict how a particular area is going to develop. When you make RFP's very strict, you may rule out getting improvements, and if requirements are not strict enough, you might not be able to tell what you are getting.

MEMBER OF THE AUDIENCE: I think this is a fundamental question, and we could almost have a session on writing the specifications for documentation. How do you describe what you are going to get and what you want?

PANEL MEMBER: A comment was made that in going through and testing these programs, operating instructions in effect sometimes are at fault. I was wondering whether the program is being run to get the same answers out that are given in the test deck without really understanding the routine itself. Maybe the other areas of documentation are just as bad to the person who is trying to find out how he can use a program and not just duplicate a set of answers.

PANEL MEMBER: At COSMIC, we do not run the program before we send it out. We compile them when we get an order for a program. Our procedure is to subcheck the program. We run a program on the program that is submitted, and this tells us all nonstandard routines that are called if they are not within the program deck. We know that it is not available and that we have to get that for the user if it is not a system program. Most of our customer complaints have been about this.

MEMBER OF THE AUDIENCE: You do not have to run a FORTRAN program that was given to you on a CDC machine and then run it against an IBM compiler. It must be against the kind of machine that it says it runs on. You do not know whether it is written in a standard language and is useful on some other machine?

PANEL MEMBER: No.

MEMBER OF THE AUDIENCE: It seems to me that the Bureau of Standards has been relatively inactive in the area of documentation standards. Because of that, everyone develops its own standards. I would like to find out to what extent Government agencies have worked with the Bureau of Standards either to help fund or cooperate in an effort to get a set of standards. It seems to me that the problem is that there are too many standards.

The other point I would like to make is that I believe that if there are standards there should be standards for different types of programs and for different complexities of programs. Different levels of documentation requirements, depending on the complexity of the programming system being developed, might be needed. In any event, it would appear to me that the Bureau of Standards should be more involved in this. They have certainly been involved in standards of programming languages.

MEMBER OF THE AUDIENCE: The National Bureau of Standards does not receive many funds in the area of documentation. One of the problems is that the Bureau must have a means of measuring conformance to a standard when it develops one. In the documentation area, this is a very subjective thing. We have not seen a set of standards that has been evaluated for effectiveness. There is no way of determining whether a particular set of standards is being promulgated by management without the concurrence of the people who have to use it. The Bureau has done a survey that shows that this might be the case in some of the standards that have been sent to us.

MEMBER OF THE AUDIENCE: Goddard Space Flight Center has been working on a set of standards for approximately 2 years. We presented a set about a year ago to NASA Headquarters. This has been distributed throughout NASA, and we hope that as a result of the comments coming back we will come up with a viable set of documentation standards. Just as some of the panel members mentioned, we started out making this extremely detailed. We felt at one time that it was quite worthwhile having it that way. At present, it has various levels of documentation, depending on the program utility, life, and size and the money spent on it. We hope to hear soon about the fate of these proposed standards.

MEMBER OF THE AUDIENCE: In 1967, the American Nuclear Society Math and Computation Division put out the *American Nuclear Society Standard for Documentation of Computer Programs,* the main intent of which was to help the exchange of computer programs. This standard was very broad. The Atomic Energy Commission has taken this document and some of the programs and has used it as a basis for their contractors in this program. They are continuing this work in a more detailed manner. We have found that we have at least started to solve some of our problems.

PANEL MEMBER: One of our problems may be that we do have standards but do not enforce the ones we have. Many Government agencies state in their contracts that documentation standards must be met. Yet these standards are not enforced by agencies. The best standard will not be at all useful if it is not enforced.

MEMBER OF THE AUDIENCE: There is another problem in developing documentation standards. Many organizations in the business of documenting do not introduce their own methods into their own organization for review. One of the real problems is that the people who are using and teaching a system do not get into an organization like the American National Standards Institute (ANSI), the Federal Users Group, or a group that will decide what a good system is for their purposes. There is some hope that an ANSI committee will be established to investigate the possibility of establishing a standard abstract, but I think those who are making a living in software have not been interested enough to cooperate.

PANEL MEMBER: In flowchart programs, do people follow more or less similar standards?

MEMBER OF THE AUDIENCE: I wonder whether the panel would comment on some of the problems that are associated with the fact that the documentation is usually received when a program is delivered. Often you cannot evaluate documentation before a contract ends. Someone also mentioned maintenance of programming. I wonder if there are any comments about how to determine whether the standards are met. In the final analysis what is important is how well the standards used are enforced.

PANEL MEMBER: I would like to say from the viewpoint of a contractor that we provide the documentation a customer asks for. Six or eight months later, we usually get the maintenance contract. We look at our own documentation rather than what we have given the customer because we provide the customer with what he wants for his purposes, which are different from our purposes. Because much of the work we do is related, we have no trouble finding someone who understands the program to re-solve the problem at the conceptual level. We do not really have as much of a problem as you would have if we delivered you a program because we are closer to it. We developed it, and we know all about it. Documentation delivered to a customer is ordinarily more a teaching aid than anything else. You have to explain to the customer the thought processes you went through so that he can learn what you have learned the hard way a little bit more easily. When we write programs, we check our programs out immediately. The computer corrects us right away. Documentation is a single-flow process, and it is much more difficult than the question-and-answer process used to write a program. That is one of the main reasons we are having so much difficulty with documentation.

PANEL MEMBER: You have seen the outline of documentation requested by COSMIC. If that was followed by your organization, do you think you could run a program that followed this guideline?

MEMBER OF THE AUDIENCE: I would think so. It is really what you would want if you were given a program to run. Whenever Goddard Space Flight Center gets a program, somebody always wants to change it, which means there will be documentation problems. People have to annotate certain parts of their program to show what they are doing.

MEMBER OF THE AUDIENCE: Are we requesting documentation that will do us any good from your viewpoint?

PANEL MEMBER: Speaking of the COSMIC standards, I think so. I have been familiar with them for a while and they seem to state quite succinctly what the customer's needs are. I do not think anybody would have any problem conforming to them. I think that they give the eventual user for whom the system is designed what he needs.

MEMBER OF THE AUDIENCE: What about someone else who is trying to use that program?

PANEL MEMBER: I do not think so. I think they are documented for a single purpose usually.

PANEL MEMBER: I would like to draw an analogy that supports this. When you build hardware and documentation for it, you generally do not expect the customer to use the system for a purpose other than the one for which it was designed. You provide the information that is sufficient to let him use the tool you have provided. The temptation in the case of software is to adapt it because everybody understands software wants to "improve" the program. A far greater depth of documentation is needed to change a program than is needed to use it or even to understand it. In this case, different levels of documentation are needed. That is another problem of documentation.

PANEL MEMBER: Perhaps you have read some of these specifications that have come out. What do you suggest doing to change these to make them useful?

MEMBER OF THE AUDIENCE: Personally, I would like to see the maintenance stay with the contractor. It solves a lot of problems, but not all of them.

MEMBER OF THE AUDIENCE: I want to know how you can insure that the documentation delivered is actually a statement of the job.

PANEL MEMBER: We have the same problem in-house that you have mentioned. We are starting now to determine what is happening using our compiler and memory allocators, and we feel that they provide as tight a code as you can get by hand. We are beginning to implement higher order language presentations or representation of the problem on disk that are available to everyone. Every change goes on the disk. Every particular simulation, whether it is scientific simulation or the fixed or fractional point simulation, is on the disk, and every step of the way we have a permanent written record that comes out. This way, since we have a hands-off process from the validated simulation, we know that what is in the machine is what we have written in the higher order language.

PANEL MEMBER: Obviously, one way to guarantee that the documentation answers what you have in hand is to have an after-the-fact documentation system. Such a system is costly for the user, but the only way that you are going to guarantee that no changes have been made up to the time that you have your document and the program is developed is to have a system that will document what you have in hand.

MEMBER OF THE AUDIENCE: I think that maintenance is very important, and I think it does belong with the original developer, just as the maintenance of hardware generally stays with the hardware manufacturer.

I would like to make the point that the formal testing of programs is an ad hoc procedure. There is no formal procedure for testing, and the history of what tests were made, when they were made, and how they were made is usually not included in the documentation.

All of this becomes very critical when you maintain and make changes to programs. For instance, the COSMIC documentation requirements talk about sample input and output. That shows what the program might do but it really does not list all the tests that were made, when they were made, and in what order they were made. I think that this is really just as important as something that tells you what the program is supposed to do when a program is being maintained. I think that there really has not been enough concentration on formal testing procedures.

PANEL MEMBER: Testing is a subject that I have been involved with for about 10 years now. The United States Air Force Satellite Control Facility has tests dating back about 7 years. Programs are very well documented when we get a program turned over that is based on a previous program. We can take a test out of our file and run the program and know that the program has at least the integrity it did before in the areas it is supposed to. We have found that we need to use very experienced people for test design. The kind of library we have is invaluable when you are working in this area, but you can only build a library like that if you are going to work in a stable system.

MEMBER OF THE AUDIENCE: We all have been assuming for a day and a half that the written code has to be unintelligible and has to be heavily documented. There are languages that force or encourage the use of short function definitions that are then combined; for example, APL. Does anyone have any comment on the advisability of reexamining the programming languages, possibly supporting changes in programming languages so that the resulting code is clearer and therefore easier to document?

MEMBER OF THE AUDIENCE: I would like to concur, I think. We have a compiler that is language independent. We have written code in FORTRAN and JOVIAL style languages. My personal feeling is that if you write clean programming and you present it to a programmer with a reasonable number of comments, he can follow through with little trouble. If it is written in assembly language, it is a little bit harder to follow because of the bulk of the data. In fact, the whole concept of understanding a program quickly is to write it in a language that the reader can read and that does not require him to read too much at one time. That is why APL is so good. On the other hand, we have had to solve that problem on our on-line terminals. We solved it by using a standard subroutine. When a new programmer begins work, he is given a copy of the subroutines and what they do and told to use them when he writes a program. In essence, we have built our own small higher order language on an on-line terminal system, and it works very well.

MEMBER OF THE AUDIENCE: The ANSI group is trying to update FORTRAN, and one of the areas that I am looking into is being able to augment the language so that it can become a better document. There has been a request to extend the character names from six to something much larger. You have to weigh what a language like that costs against what a language that is easy to learn and compile costs. The question is how much you are willing to spend to enhance the understanding of the program. I think these are the things that have to be weighed.

MEMBER OF THE AUDIENCE: It seems to me that we have standards on one hand and guidelines on the other. Many of the papers this morning have presented not standards but guidelines. The person preparing a contract can decide which of these guidelines should actually be made standards for his contract..

PANEL MEMBER: Let me suggest that we might loosen the standards we have a little bit and let the machine do some of the documentation. Maybe there has been a little too much emphasis on standards. There are a lot of languages now that provide a lot more capabilities. Why not start removing some of the requirements and let the machine do more of the work?

PANEL MEMBER: I would like to say that the person or the group that is most likely to have its standards accepted by a community is going to be the one that has contributed the most to that community. I suspect that we are waiting for one or two more software inventions that will solve the documentation problem by supplying a convenient way of doing it.

PANEL MEMBER: I think any automatic system could be tailored to a format that people will want, and I think Goetz' system is flexible enough to supply the type of documentation that you need for a particular purpose. If more effort is put into this field, automated systems that provide the various levels of documentation needed can be developed.

MEMBER OF THE AUDIENCE: One project is trying to automate abstracts across communications lines. The people working on this are essentially library people who are not the ones who will use the data.

MEMBER OF THE AUDIENCE: I would like to change the subject slightly. There seems to be a marked tendency lately for the Government to want to acquire unlimited data rights. The tendency of many Government contracting officers to feel that the Government has free right to any aspect of anything used on a Government contract seems to be connected with this. I was wondering whether this will foster or hinder the development of automated data documentation systems.

PANEL MEMBER: I think it can work both ways. I think very frankly that any company considers certain things proprietary. If we used our compiler, which is machine independent, target computer independent, and language independent on a Government contract, before we even started we would say that it is something we cannot give away. We have invested a lot of money in it, it is a very powerful tool, and we think it will keep us ahead of our competitors for a few years.

Automated documentation has one major advantage—it is predictable. If something is predictable, you know how to write your programs to satisfy the requirements without necessarily giving away proprietary rights. Automatic documentation satisfies all requirements. It satisfies the customer's demands and the recognition of certain property rights in programs that are recognized as belonging to the originator.

PANEL MEMBER: Precedents have been set for the Government either to lease with a limited right or to buy with a limited right. If you do have a package that you were going to use in a project, you would probably be willing to either lease or sell it for a specific use.

MEMBER OF THE AUDIENCE: I have heard a lot of discussion about documenting systems and programs within a system and functional documentation of programs. I would like to ask whether we need to document programs for modification or just for maintenance, and when should they be so documented? For example, COSMIC suggests that the program would go to a user and be modified by the user. The kind of documentation needed for this is considerably different from that needed for maintenance of a system.

PANEL MEMBER: I think that the only solution to documentation is to get the 90 percent of the documentation that can be done by computer and the other 10 percent done by the people that have it in their head and then to figure out a way to integrate the two. I agree with the point that documentation is always looked at at the very end. For one thing, most of the money is already paid to the contractor at that time. Secondly, you really do not have time to do anything about it, and the documentation really should have been in phase with the development work, which is generally not a requirement.

PANEL MEMBER: My own observation about Goddard Space Flight Center is that what you are describing does happen on the big projects. Documentation is reviewed on them from step to step. I think that when the project is very important and the money is available, documentation is taken care of more conscientiously. It is on the intermediate projects when the budget is tight and we are trying to get something done as quickly as possible that we run into more trouble. In this case documentation is done after the project is completed.

MEMBER OF THE AUDIENCE: One of the advantages of Bellflow is that in the mixed mode the comments have to agree with the source. If you are developing a program and produce a flowchart and the comments agree with the source, then you get a flowchart. If, 5 months later, someone changes the source but does not change the comments and you try to get the flowchart again, you get a diagnostic. This is one way we can automate and test with documentation.

Session IV

N73-19220

PROGRAM AUTOMATED DOCUMENTATION METHODS

Bernadine C. Lanzano
TRW Systems Group

Several methods for automatically generating and maintaining documentation for TRW's computer programs are being used, and other procedures are under examination. This paper presents a short synopsis of the Mission Analysis and Trajectory Simulation (MATS) program to provide an understanding of the size and complexity of one simulation for which documentation is mandatory, a description of a program that assists in automating the documentation of subroutines, an exposition of two flowcharting programs, some notes on useful program internal cross-reference information, an implementation of a text-editing program available in a time-shared computer system environment, a preview of a proposed system that would aid in program development and documentation that utilizes a graphics display console, and a recommendation for software standardization.

In the complex world of sophisticated computer systems and advanced software technology, Thompson Ramo Wooldridge, Inc. (TRW), has long recognized the need for developing general-purpose programs, automating documentation, and standardizing programming techniques. The satisfaction of these demands minimizes software expenses by eliminating program duplication, developing new capabilities around and within existing programs, responding in a quick reaction real-time sense, and by generating documentation with minimum effort.

The MATS program is briefly mentioned because its generality, complexity, and size necessitate considerable support documentation. Because this program is used by a variety of projects, its documentation is referenced by many engineers, programmer/analysts, and technical aids. Methods of automating the documentation were deemed mandatory.

MATS is a digital computer program that simulates ballistic and space mission trajectories; it either has or is capable of simulating such missions as Apollo, Pioneer, Minuteman, and Grand Tour, with such vehicle configurations as Saturn, Atlas, Titan, Agena, Centaur, Minuteman, and Thor, among others. The program is written in FORTRAN IV and is operable on the CDC 6000, GE 635, IBM 360, and IBM 7094 computer systems. MATS is composed of more than 450 subroutines that occupy some 110 000 decimal words in 15 overlaid segments. The program control logic is predicated on a modular design concept that facilitates the addition or exchange of capabilities for the various missions. It can be mated with control systems that include navigation and guidance algorithms and can provide the dynamics for interpretive computer simulation systems.

For any program, several levels of documentation are required. The smallest unit is the subroutine where the function, algorithms, and data communication must be explained. A lineal linkage trace of the program logic control hierarchy provides the next level of documentation. Two-dimensional cross-reference information is desirable so that not only which subroutine(s) and common storage(s) are used by a given routine are known but also which routines reference this routine and which routines reference each element of global storage.

The descriptive material that functionally relates the program modules can be generated only by the program architect and must be updated as the program expands and evolves. Last but probably most important are the user manuals, which again must be generated in an easily maintainable format.

This paper describes several auxiliary programs that support the automatic documentation of MATS and other programs.

(1) Automated Documentation of Subroutines (ADS) mechanizes the descriptive explanation at the subroutine level.
(2) Flowcharters AUTOFLOW and FLOWGEN pictorialize the computational algorithms and decision branches within a subroutine. AUTOFLOW can be requested to provide a higher level flowchart.
(3) Automatic Flow Layout of Program (AFLOP) maps the subroutine usage at the program level.
(4) Methods for retrieving and programs for generating the cross-reference information are presented.
(5) Administrative Terminal System (ATS) assists in automating the maintenance of handbooks and manuals.
(6) Computer-Aided Program Development (CAPD) proposes a method wherein the coding and final flowcharting no longer appear as steps in program development and documentation.

One of the purposes of this paper is to demonstrate how an individual piece of information can be made to function in more than one capacity, thus deleting duplication of effort in creating and maintaining documentation files.

AUTOMATED DOCUMENTATION OF SUBROUTINES

This program, currently under development, accepts as inputs FORTRAN subroutine source decks, a subroutine titles file, and a symbols definitions file. An option to retrieve storage and external reference information from the compiler output or the list tape is under review.

The source deck contains the following information, where Cnn is the appropriate coded comment card: the title, a one-line title that normally spells out the name of the subroutine (C10); the author, the programmer/analyst, and other personnel cognizant of the function and/or implementation of the subroutine (C30); the abstract, a meaningful description of the purpose and function of the subroutine (C40); remarks, information that briefly describes any change and gives the analyst's name and date and any special features, restrictions, limitations, and error treatments (C50); and the local variables, the names and definitions of those variables that are used only within the subroutine including those in the calling sequence (C60).

The subroutine title file is merely the title (C10) cards identified with their respective subroutine names.

The symbols definition file contains the name, common block location, definition, and units of every global variable used in the program.

These cards are formatted in the following manner:

```
C EQU (VARNAM, COMBLK(nnnn))   DIM(iii,jjj)  IO  UNITS=kkkk      VARNAM  00
        DEFINITION (on as many cards as required)                VARNAM  mm
```

where EQU implies equivalence; VARNAM is the variable name; COMBLK(nnnn) is the name of the common array and location of the variable within the block; DIM(iii,jjj) is the dimension of the variable; I or O indicates whether it is a user input or a program computed variable; UNITS = kkkk provides information pertinent to the variable, such as length and time units, integer or real format, and special processing information; and DEFINITION continues from one card to the next and is identified by the variable name and card count mm in the final columns.

A parenthetical explanation of the symbols definitions file is in order. The MATS program requires the information provided by the EQU cards for the symbol table, input, computations, and output processors to locate and identify each variable so that all input and output may be recognized by symbol name. The symbol table processor accepts as input the EQU . . . 00 file, alters the file by change directives, and outputs an updated file. The symbol definitions file, using standard sorting equipment, is updated and published as a portion of the MATS users manual. This file, being carefully designed and formatted, thus may be used in three distinct areas: the MATS program, the ADS program, and the MATS documentation. Furthermore, it is easily and automatically updated.

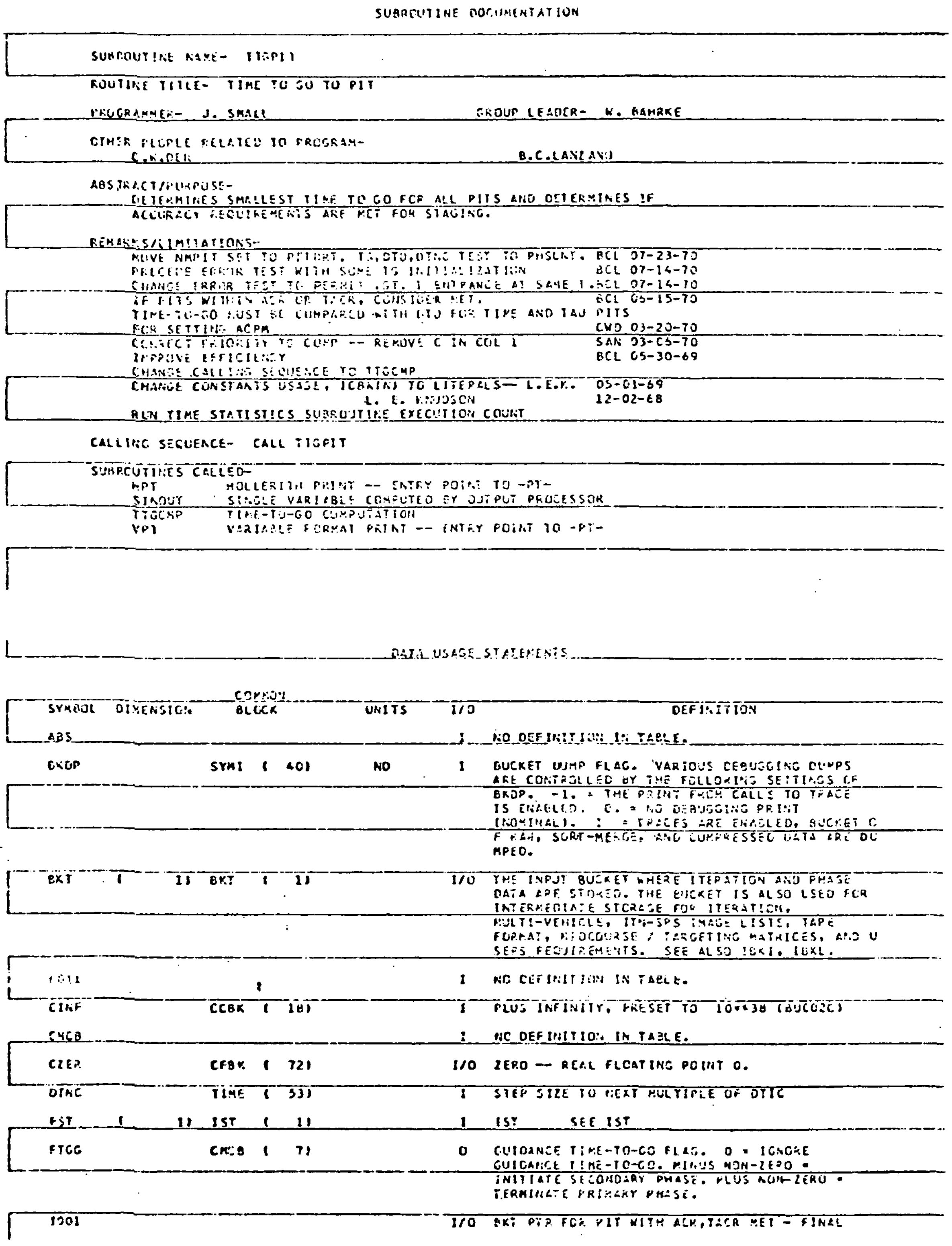
The ADS program searches the source deck from the subroutine card to the end card. It retrieves the name from the subroutine card and the title, author, abstract, and update information from the coded comment cards. It identifies the external references and acquires their titles from the titles file. It identifies the variables and locates their definitions either from the symbol definitions file or from the C60 cards. Figure 1 presents the output of the ADS program.

The appearance of the coded comment cards in the MATS program listing proves very useful to the programmer/analyst who, when working with the subroutine, finds the associated documentation immediately available.

TRW has other programs similar to ADS that operate on the IBM 7094 and IBM 360 to document programs that execute on those computer systems. Each is clearly designed to automate subroutine documentation with minimum manual effort.

FLOWCHARTING PROGRAMS

Two flowcharting programs are currently in use at TRW: AUTOFLOW and FLOWGEN. Each is leased from the respective vendor. They are used primarily for the charting of individual subroutines, but AUTOFLOW can produce charts for a complete (small) program. The symbology of both is nearly self-explanatory and quite similar to that commonly used by analysts.



```
                                SUBROUTINE DOCUMENTATION

        SUBROUTINE NAME-  TTGPIT

        ROUTINE TITLE-  TIME TO GO TO PIT

        PROGRAMMER-  J. SMALL                            GROUP LEADER-  W. BAHRKE

        OTHER PEOPLE RELATED TO PROGRAM-
             C.W.DEN                                          B.C.LANZANO

        ABSTRACT/PURPOSE-
             DETERMINES SMALLEST TIME TO GO FOR ALL PITS AND DETERMINES IF
             ACCURACY REQUIREMENTS ARE MET FOR STAGING.

        REMARKS/LIMITATIONS-
             MOVE NMPIT SET TO PITBRT. TS,DTO,DTNC TEST TO PHSCNT. BCL 07-23-70
             PRECEDE ERROR TEST WITH SOME TS INITIALIZATION        BCL 07-14-70
             CHANGE ERROR TEST TO PERMIT .GT. 1 ENTRANCE AT SAME T.BCL 07-14-70
             IF PITS WITHIN ACR OR TACR, CONSIDER MET.             BCL 05-15-70
             TIME-TO-GO MUST BE COMPARED WITH DTO FOR TIME AND TAU PITS
             FOR SETTING ACPM                                      CWD 03-20-70
             CORRECT PRIORITY TO COMP -- REMOVE C IN COL 1         SAN 03-05-70
             IMPROVE EFFICIENCY                                    BCL 05-30-69
             CHANGE CALLING SEQUENCE TO TTGCMP
             CHANGE CONSTANTS USAGE, ICBKIN) TO LITERALS— L.E.K.  05-01-69
                                        L. E. KNUDSON             12-02-68
             RUN TIME STATISTICS SUBROUTINE EXECUTION COUNT

        CALLING SEQUENCE-  CALL TTGPIT

        SUBROUTINES CALLED-
             HPT        HOLLERITH PRINT -- ENTRY POINT TO -PT-
             SINOUT     SINGLE VARIABLE COMPUTED BY OUTPUT PROCESSOR
             TTGCMP     TIME-TO-GO COMPUTATION
             VPT        VARIABLE FORMAT PRINT -- ENTRY POINT TO -PT-



                                  DATA USAGE STATEMENTS

                         COMMON
    SYMBOL  DIMENSION    BLOCK           UNITS     I/O                    DEFINITION

    ABS                                              I   NO DEFINITION IN TABLE.

    BKDP                 SYMI  (  40)     NO         I   BUCKET DUMP FLAG. VARIOUS DEBUGGING DUMPS
                                                         ARE CONTROLLED BY THE FOLLOWING SETTINGS OF
                                                         BKDP.  -1. = THE PRINT FROM CALLS TO TRACE
                                                         IS ENABLED.  0. = NO DEBUGGING PRINT
                                                         (NOMINAL).  1  = TRACES ARE ENABLED, BUCKET O
                                                         F RAW, SORT-MERGE, AND COMPRESSED DATA ARE DU
                                                         MPED.

    BKT      (      1)   BKT   (   1)              I/O   THE INPUT BUCKET WHERE ITERATION AND PHASE
                                                         DATA ARE STORED. THE BUCKET IS ALSO USED FOR
                                                         INTERMEDIATE STORAGE FOR ITERATION,
                                                         MULTI-VEHICLES, ITN-SPS IMAGE LISTS, TAPE
                                                         FORMAT, MIDCOURSE / TARGETING MATRICES, AND U
                                                         SERS REQUIREMENTS.  SEE ALSO IBKI, IBKL.

    COIL                                             I   NO DEFINITION IN TABLE.

    CINF                 CCBK  (  18)                I   PLUS INFINITY, PRESET TO  10**38 (BUC02C)

    CMCB                                             I   NO DEFINITION IN TABLE.

    CZER                 CFBK  (  72)              I/O   ZERO -- REAL FLOATING POINT 0.

    DTNC                 TIME  (  53)                I   STEP SIZE TO NEXT MULTIPLE OF DTIC

    FST      (      1)   IST   (   1)                I   IST       SEE IST

    FTGG                 CMCB  (   7)                O   GUIDANCE TIME-TO-GO FLAG.  0 = IGNORE
                                                         GUIDANCE TIME-TO-GO. MINUS NON-ZERO =
                                                         INITIATE SECONDARY PHASE. PLUS NON-ZERO =
                                                         TERMINATE PRIMARY PHASE.

    I901                                           I/O   BKT PTR FOR PIT WITH ACR,TACR MET - FINAL
```

Figure 1.–Example of ADS output.

AUTOFLOW accepts COBOL, FORTRAN, IBM 360 Assembly, and PL/I source programs as inputs. The AUTOFLOW option generates a chart set composed of the title sheet, input listing, statement label index, table of contents, table of diagnostics, flowcharts, and other special listings; some of these items are optional. It charts an entire program up to 999 flowchart pages.

The CHART option operates from specially coded comment cards that may be embedded in the program source deck and produces a higher level program chart from the textual information. The author may adjust the level of detail to the type of chart he wishes to exhibit.

AUTOFLOW executes on the IBM 360; each chart page covers two 11- X 17-in. printer pages and may contain up to four columns of paths. Pages and symbols are numbered to facilitate page-wide logic flows. Figure 2 exhibits an AUTOFLOW chart.

FLOWGEN accepts FORTRAN source decks as inputs and outputs a chart somewhat less sophisticated than AUTOFLOW. No provision exists for a level of detail control. It charts individual subroutines.

FLOWGEN executes on the CDC 6000 and generates input for the CALCOMP plotter either directly or on tape. Each chart page is 8.5 X 11 in. with one column of flow path. Pages are numbered, and symbols are supplied to chart page-broken logic flow paths. Figure 3 depicts a FLOWGEN chart.

Both flowcharters completely automate the charting of programs. However, most analysts will concede that manually manipulated page topology is generally more acceptable than automated columnized formats, particularly for large, complex subroutines.

AUTOMATED DOCUMENTATION OF PROGRAM INTERNAL COMMUNICATION

Given that a program is composed of a collection of subroutines where the word "subroutine" is a generic term including functions, entry points, block data, and other subelements, certain program internal intersubroutine communication documentation is desirable. Useful cross-reference information would include forward reference, backward reference, and flow hierarchy. Forward references include–

(1) All subroutines referenced by this subroutine
(2) All commons referenced by this subroutine
(3) All global variables defined in common arrays referenced by this subroutine, and
(4) All local variables defined within and referenced only by this subroutine

Reverse references include–

(1) All subroutines that reference this subroutine
(2) All subroutines that reference this common, and
(3) All subroutines that reference this global variable

Flow hierarchy is the cascade of subroutine forward references that presents an overall view of the program flow logic.

Table 1 summarizes those portions of the cross-reference information that are available from the manufacturers' standard software systems: the CDC 6000 compilers and overlay

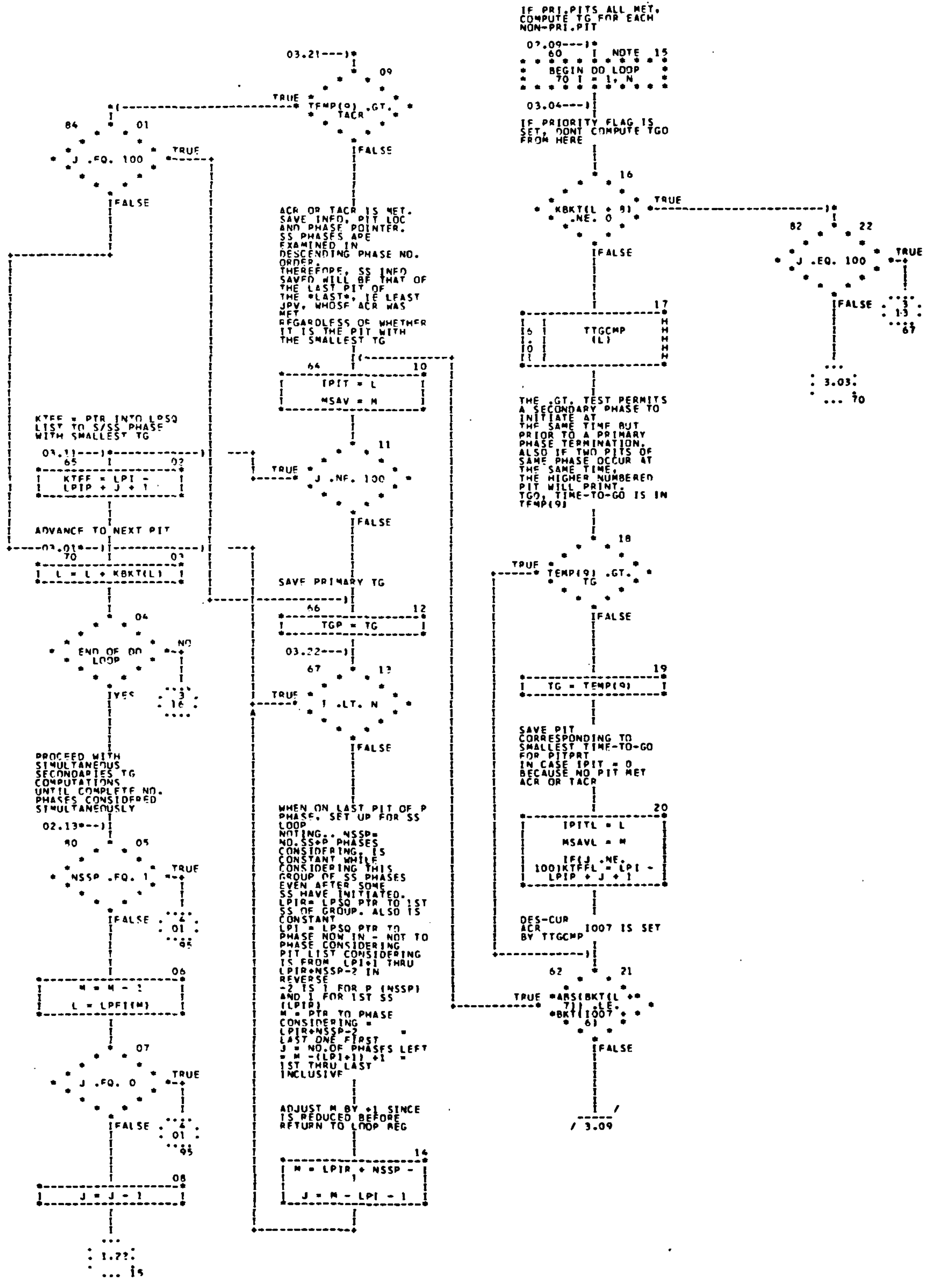
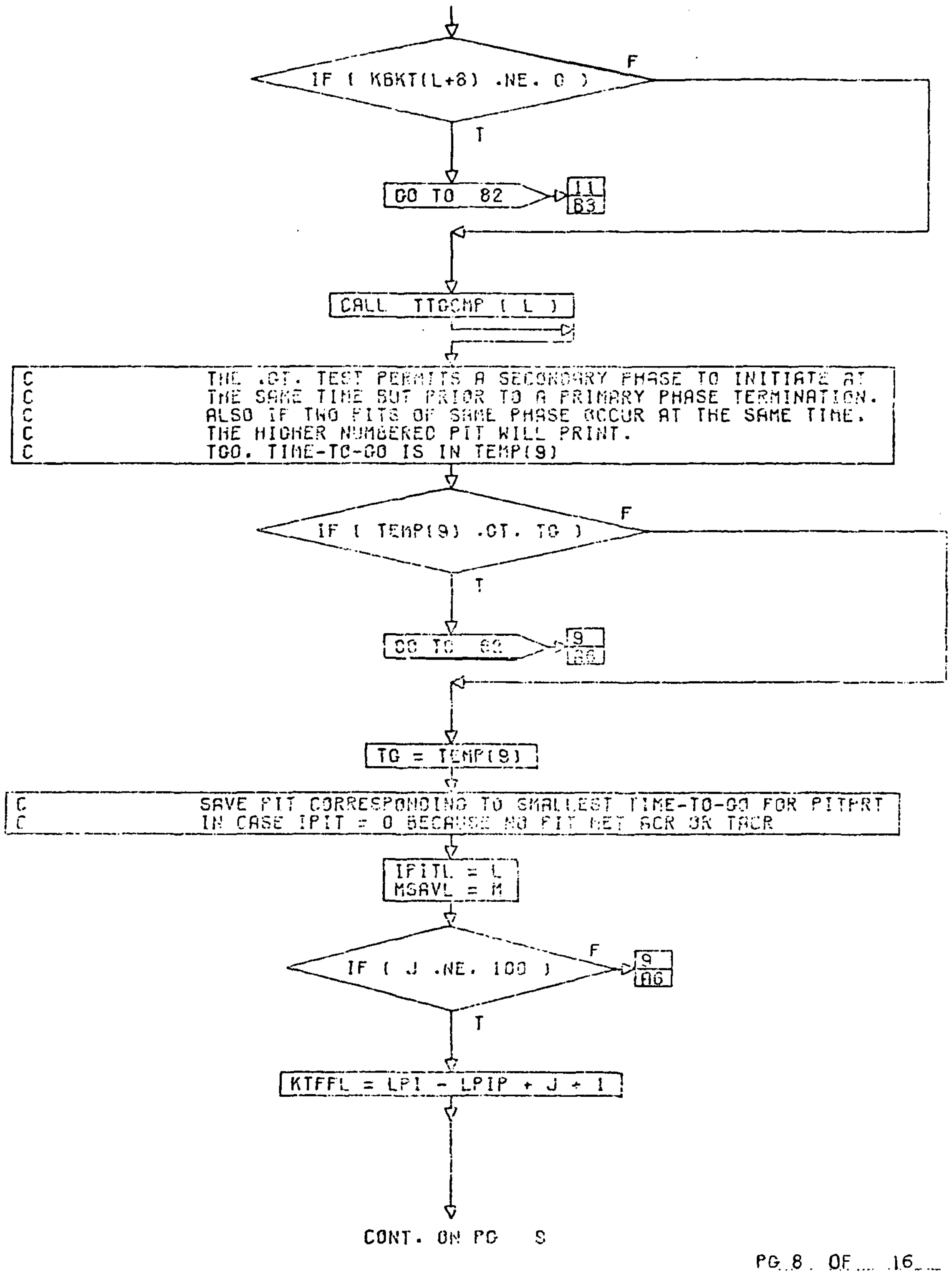

IF PRI.PITS ALL MET, COMPUTE TG FOR EACH NON-PRI.PIT
BEGIN DO LOOP 70 I = 1, N
IF PRIORITY FLAG IS SET, DONT COMPUTE TGO FROM HERE
KBKT(L + 9) .NE. 0
J .EQ. 100
TTGCMP (L)
TEMP(9) .GT. TACR
ACR OR TACR IS MET. SAVE INFO, PIT LOC AND PHASE POINTER.
IPIT = L
MSAV = M
J .NE. 100
SAVE PRIMARY TG
TGP = TG
I .LT. N
KTEE = LPI - LPIP + J + 1
ADVANCE TO NEXT PIT
L = L + KBKT(L)
END OF DO LOOP
NSSP .EQ. 1
M = M - 1
L = LPFI(M)
J .EQ. 0
J = J - 1
TEMP(9) .GT. TG
TG = TEMP(9)
IPITL = L
MSAVL = M
ABS(BKT(L + 7)) .LE. BKT(1007 + 6)
ADJUST M BY +1 SINCE IS REDUCED BEFORE RETURN TO LOOP BEG
M = LPIR + NSSP - 1
J = M - LPI - 1
TRUE
FALSE

Figure 2.–Example of AUTOFLOW flowchart.



```
C                    THE .GT. TEST PERMITS A SECONDARY PHASE TO INITIATE AT
C                    THE SAME TIME BUT PRIOR TO A PRIMARY PHASE TERMINATION.
C                    ALSO IF TWO PITS OF SAME PHASE OCCUR AT THE SAME TIME.
C                    THE HIGHER NUMBERED PIT WILL PRINT.
C                    TOO. TIME-TO-GO IS IN TEMP(9)
```

```
C                    SAVE PIT CORRESPONDING TO SMALLEST TIME-TO-GO FOR PITPRT
C                    IN CASE IPIT = 0 BECAUSE NO PIT MET ACR OR TACR
```

Figure 3.–Example of FLOWGEN flowchart.

Table 1.—Program Cross-Reference Information Available From Standard System Software

Software system	Forward reference function				Reverse reference function		
	1	2	3	4	1	2	3
Compilers:							
CDC RUN	No[a]	Yes	Yes	Yes	([b])	([b])	([b])
CDC FUN	No	Yes	Yes	Yes	([b])	([b])	([b])
CDC FTN version 3	Yes	Yes	No[c]	No[c]	([b])	([b])	([b])
GE GECOS	Yes	Yes	No	No	([b])	([b])	([b])
IBM 360	Yes	Yes	Yes	Yes	([b])	([b])	([b])
IBM 7094	Yes	Yes	Yes	Yes	([b])	([b])	([b])
IBM 7094 assembler	Yes	Yes	No	Yes	([b])	([b])	([b])
Loaders:							
CDC loader map	No	Yes	No[b]	No[b]	Yes[d]	No	No
GE loader map	Yes	Yes	No[b]	No[b]	No	No	No
IBM 360 linkage editor	No[e]	No[e]	No[b]	No[b]	No	No	No
IBM 7094 load map	No	Yes	No[b]	No[b]	No	No	No
IBM 7094 logic map	Yes	Yes	No[b]	No[b]	Yes	Yes	No
NASTRAN linkage editor	Yes	No	No[b]	No[b]	Yes	No	No

[a]TRW has modified the CDC RUN compiler to output the referenced subroutines.
[b]Not applicable for the compilers and perhaps desirable from the loader maps only as an option.
[c]An option causes the local variables to be printed and the locations of all references to a common array to be listed; it does not print the names of the global variables.
[d]The subroutine reverse references are available only within an individual program overlay.
[e]An option generates the name of the referenced subroutine and/or common along with the locations, not names, of the referencing subroutine.

loader; the IBM 360/50/65/85 levels G and H compilers and linkage editor; the IBM 7094 IBSYS compiler, assembler, and IBLDR loader; and the NASTRAN loader, which operates like the IBM 360 linkage editor on the CDC 6000 computer.

As can be seen, five of the seven compilers mentioned generate the subroutines referenced by a given subroutine. All the compilers give the referenced commons. Four compilers list the global variables, and five list the local variables referenced by the subroutine. Three of the six loaders give forward subroutine references, and four give the commons referenced by a subroutine. No loader names a variable. This would be desirable from the loader map as an option, but the variable names are probably not immediately available.

The reverse references for subroutines are for all practical purposes missing from four of the loaders, and only one generates all references to a given common. Nowhere is information available to yield all references to a global variable, which is highly desirable in validating and maintaining a program.

No standard loader generates a complete expansion of the forward subroutine references, which also is desirable in using a program.

To surmount these deficiencies and to provide what is deemed useful documentation, TRW has developed several programs. Figures 4 and 5 represent their outputs.

Symbol Do. Name
Name

	LKZ	DLK	DTA	CLT	UTL	OPT	ERR	INT	IPT	CRD	PRT	PRM	FRN	PRO	PRP	PRQ	PRR	BSF	COM	FST	RSO	NRS	CHK	TRT	TTI
00ERR					U		B																		
06ERR1							B																		
06ERRA							B																		
06ERR					U		D																		
08ERR					U		D																		
09ERRC							B																		
09ERRN							B																		
09ERRR							B																		
09ERR					U		B																		
09ERRS							B																		
0BOFF			D																	U					
0BTYP																				B					
0L1			D					U																	
0L2			D																						
0L3			D													U									
0MP			B		U					U		U						U	U	U					
0MR			B									U							U	U					
0MY			B									U							U	U					
OPTION			D		U		U	U		U	U	U	U	U		U				U	U	U	U	U	
ORDER			D					U												U				U	
106RK												B			U										
106RKZ												B													
10AZZA									B																
10AZZB									B																
10AZZC									B																
10AZZ									B																
10AZZX									B																
10ERR					U		D																		
11ERR							D																		
12X4									B																
16ERR					U		D																		
17ERR					U		D																		
19ERR					U		D																		
1AAZZB									B																
1AAZZ									B																
1ACGC													D			U								U	
1ACGCZ													B												
1ACTD1					B																				
1ACTD2					B																				

Figure 4.–Example of global variable cross-reference, IBM 7094 program.

```
SYMBOL        SUBROUTINE

A             NVRS3D    TVGNL     TVGSC
AA            NVRS3D
AAP           AANGT     APRIT     RCNFT
AAPM          AANGT
AAT           AANGT     APRIT     HCNFT
AAY           AANGT     APRIT     RCNFT
AAYM          AANGT
ACSOR         CREAD     INPROC
ACTNMR        CRFAD     INPROC
AERC          AATMP     AFFDR     AIRVEL    APRIT     DCNFT     BLKDTA    CFEAD     ARBIT
              MDRE      MDRAG     SGFFL
              RTAMP
AHT           TM1PS     T1AFF     TM1NS     TNAFF     T2AFF     T3AFF     TVGCD     TVGMC
              TVGSC
AIRSHP        CREAD     INPROC
ALATL         ASTART    CREAD     INPROC    INPRUC
ALMCOF        TVGMC     TVGNL
ALTGT         INPROC
AMUL          ASTART
AMUT          BIMAZ     TM1PS     TM1NS     T3AFF     TCCTD     TVGMC
AMXLOO        MCDER
APAF          AATMF     AATMP     ADVIC     AFFDR     AIRVEL    APFDR     APRIT     ASPAC
              ATGOE     BCNFT     BLKDTA    MDRAG     MTHWF     SGFFL
APDPT         TVGNL
APDR          TVGNL
APSI          AIRVEL    APFDR     ATMOS     MTHWF
ASTRH1        BSTRS3    SGBST
ASTRH2        BSTRS3    SGBST
ATEMP         AIRVEL    ATMOS     NVRS3D
AZCORR        ASTART    INPROC
AZL           APRIT     ARBOR     ASTART    AZEST     CREAD     ARBIT     T3AFF
AZL1          T3AFF
AZMUT         ANIP      BIMAZ     TCCTD
AZPTS         CREAD     INPROC
AZSECT        CREAD     INPROC
B             NVRS3D    TVGNL     TVGSC
BB            NVRS3D
BBARCO        TVGCD     TVGMC     TVGNL     TVGSC
BETA          AAUXF     APRIT     ATCRS     ATGOE     MDRAG     TM1PS     TM1NS     TMLIN
BETI          AAUXF     APRIT
BIGEST        ATGOE
BLANK         INPROC
BURST         CREAD     INPROC
BURSTA        CREAD     INPROC
BURSTB        CREAD     INPROC
C             TVGNL     TVGSC
CALM          ACDER     ANAVH     ANIP      BLKUTA    EGRAV
CASE          INPROC    ?RGFNC
CD            AATMP     AFFDR     AIRVEL    APRIT     MDRAG     SGFFL
CDDELT        MDRAG
CDMULT        AATMP     AFFDR     MDRAG     SGFFL
CDPR1         MDRAG
CDPR2         MDRAG
CDPR3         MDRAG
CDREF         MDRAG
CGLATL        AZEST     INPROC
CGM           AD2CG     SGBST
CGOFFS        AD2CG     APFDR
CGOFFT        AD2CG     APFDR
```

Figure 5.—Example of global variable cross-reference, IBM 360 program.

Figure 4 presents the output of the symbol reference program, which lists every global and local variable along with the name of the deck that defines (D), uses (U), or both defines and uses it (B). Deck here may be a collection of subroutines. Written in IBM 7094 assembly language and operating under the IBSYS system, this program documents an IBM 7094 assembly language program from its output list tape.

Figure 5 presents the output of a similar symbol reference program, which lists the global variables with every subroutine reference. This program is written in IBM 360

FORTRAN and uses as inputs the common and equivalence statements from FORTRAN subroutine source cards.

This variable reference information may also be generated for other IBM 7094 machine language programs that operate in the TRW SCAT system. This additional capability for the ADS program is currently under review for the CDC 6000; it is designed to document FORTRAN IV programs and to accept CDC list tapes and/or source cards as inputs.

Figure 6 presents the flow hierarchy of a program that is the output of AFLOP; it expands the subroutine references until it exhausts the calls or reaches an undefined external. It currently operates on the IBM 360 and the CDC 6000 computers. AFLOP accepts input cards that spell out the names of the referencing and referenced subroutines. An option permits subroutine expansion at each encounter or line number reference to the first expansion. TRW intends to incorporate AFLOP as an option of the CDC 6000 NASTRAN linkage editor and to make it available for the CDC 6000 overlay loader; in these systems it acquires its inputs from the loader tables.

ADMINISTRATIVE TERMINAL SYSTEM

Formerly, certain documents, such as programmer's handbooks and users guides, have been prepared manually. A revision usually meant considerable retyping and proofreading, both of which consumed time and could introduce errors. Several text editors have appeared on the market; a good one is the ATS developed by IBM under the acronym DATATEXT. TRW currently purchases time to use this system and is contemplating installing the program in-house.

The system uses an IBM 360 computer with appropriate storage devices and high-speed printers at the central site. Telephone lines connect the computer with the remote stations. The terminal may be an IBM 2741 or a DATEL 30, and either may be hard wired or connected to a standard telephone set with a data coupler. The keyboard resembles an IBM Selectric typewriter and may be used as such when disconnected from the computer. The operator-secretary enters a document by instructing ATS with margin placements, tab settings, and formats; the text is typed and the system is requested to file it in permanent storage. ATS assigns line numbers for subsequent editing.

Features of the system include indented and blocked paragraphs, page width and depth control, page headings and footings with automatic centering and numbering, line and page skip, margin justification, and table and chart special formats. Lines may be kept together; for example, in a table that should not be split across pages. Form letters may be prepared, and, with the stop code, one may request the printing to halt temporarily for the insertion of particular data.

The print options include printing some portion or the whole document with or without line numbers, with or without justification, at the terminal or on the high-speed printer. ATS displays its agility in the editing capabilities. Corrections reference the line number and any word within the line. One may remove or replace a word, a phrase, or a line, add to or remove lines, and physically move lines or paragraphs. The edited document as well as the original version may be retained in storage.

```
1MISCN1
   2  ADDRNT
   3          ADDRES
   4          MRELSE
   5                  CFC
   6          OUTPTC
   7                  GETBA
   8                  SYSTEM
   9                  ABNORML
  10                  OPEN.
  11                          ADVIN.
  12                  STO.
  13                  DAT.
  14                  POSFIL.
  15  CMPINC
  16          PHSCNT
  17                  PSREAD
  18                          REWINM
  19                                  SYSTEM
  20                                  ABNORML
  21                                  GETBA
  22                                  CIOI.
  23                                  ADVIN.
  24                          OUTPTC  6
  25                          KILRUN
  26                                  NAMLOC
  27                                  OUTPTC  6
  28                                  LINK
  29                          INPUTB
  30                                  GETBA
  31                                  SYSTEM
  32                                  ABNORML
  33                                  CIOI.
  34                                  STO.
  35                                  OPEN.   10
  36                                  MVADS.  *
  37                          PT
  38                                  LOCF
  39                                  OUTPTC  6
  40                          INSPRD
  41                  STAGCN


  42                          PRESET
  43                                  PHSVAN
  44                          UNPKIO
  45                          OUTPTC  6
  46                          IPT     *
  47                          KILRUN  25
  48                          PITSTS
  49                                  DATSTO
  50                                          OUTPTC  6
  51                                  UPITMD
  52                                  VARSET
  53                                  STNOUT
  54                                          ACCUFR
  55                                                  SYSTEM
  56                                                  ABNORML
  57                                          OUTSYS
  58                                                  OUTPTC  6
  59                                                  COPYDA
  60                                                  OCHOYC
  61                                                          EPHEMX
  62                                                                  INTR2
  63                                                                          FILOPN
  64                                                                                  PT      37
  65                                                                                  KILRUN  25
  66                                                                                  OUTPTC  6
  67                                                                                  PT      37
  68                                                                          IDINT
  69                                                                                  SYSTEM
  70                                                                          DBLE    *
  71                                                                          SNGL    *
  72                                                                          BACKSP
  73                                                                                  SYSTEM
  74                                                                                  ABNORML
  75                                                                                  GETBA
  76                                                                                  CIOI.
  77                                                                                  BKSPRU. *
  78                                                                                  FIZRAK.
  79                                                                                  ADVIN.
  80                                                                                  FIZRA.
  81                                                                                  POPRU.  *
  82                                                                          REWINM  18
```

Figure 6.–Example of AFLOP.

Some inherent disadvantages are the lack of superscript and subscript notation and the omission of special characters such as the Greek alphabet. The preparation of equations manuals can be only partially automated; the text could be maintained using this editor with space allowed for the manual entering of the mathematical equations or diagrams.

The ATS system provides various administrative facilities such as a log of the documents stored; the date, name, and size of each; and the total number of documents in storage. Complete or partial storage reports may be requested. Each document may contain a password that prevents anyone who does not know the password from accessing the document. The password may be changed as often as desired to achieve some level of security.

The MATS manuals are partially maintained via ATS, and it is intended that all future documentation be implemented with this system.

Brief mention should be made of other text-editing programs. One is the TRW General Trajectory Documentor (GTDOC) program. It accepts a file of prebuilt text from tape or cards, a card deck of text modifications, and a data set of trajectory parameters either from tape or cards. The output is a standard-form document with the trajectory data positioned properly in the text. GTDOC automates the preparation of trajectory-oriented publications. It operates on the IBM 7094.

Although the TRW Timeshare System Editor is designed primarily to aid in preparing executable programs, it incorporates commands useful in constructing other types of data files.

COMPUTER-AIDED PROGRAM DEVELOPMENT PROPOSAL

Occasionally, it is advantageous to analyze the procedures normally pursued in the development of a program. In addition to the normal preliminary functions of defining and specifying the usual program performance criteria, the steps involved are—

(1) Creating a flowchart
(2) Defining the flowchart
(3) Generating the program code
(4) Analyzing the coding errors
(5) Analyzing the design inadequacies
(6) Iterating (2) through (5) until complete
(7) Documenting the subroutine(s) by redrawing the flowchart(s)
(8) Developing the program further by iterating (2) through (7)

Considerable time is expended in generating a flowchart from which the programmer/analyst prepares the program, as any engineer or analyst can readily attest. Refining this flowchart to introduce even one new equation requires providing the right space at the right place (foresight), erasing and shifting the symbols with contents (copy errors), and redrawing and shifting (copy errors).

The code is then revised to match the flowchart, which often results in inefficiency in the code and a generally disorganized arrangement in both the sequence of operations and format of the subroutine. Eventually, a new flowchart is necessary. Program evolution consists of flowchart, code, analysis, flowchart.

CAPD proposes a system wherein the code and the final flowchart no longer appear as steps in program development. This technology uses a graphics display console with character, line, and vector capabilities. The IBM 2250 and the CDC Digigraphics consoles are potential candidates. Time sharing is desirable for economic purposes only.

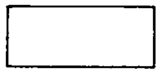
The initial flowchart is created with aid from the computer, which provides the flowchart symbols and automated spacing. Modifications are achieved with maximum ease and reliability with the CAPD graphics program. The analysis proceeds at an accelerated pace because CAPD permits the analyst to concentrate on the problem, aids in diagnosing the flow diagram, and supplies definitive information on request or on repeated error occurrences.

CAPD Translator

The translator takes input from the graphics flow diagram in terms of arithmetic and logical expressions enclosed within the flowchart symbols. It transcribes this flowchart into source language appropriate for compiler input by translating, for example, rectangles into arithmetic statements, hexagons into CALLs, and triangles and diamonds into IF statements.

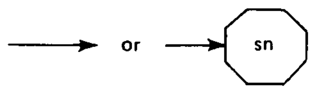
CAPD Conventions

The conventions follow FORTRAN closely and adopt common flowchart symbol graphics representations:



Arithmetic statements: rectangle. The operators +, -, *, /, **, or ↑ and the field delimiters =, ,, () carry FORTRAN definitions.

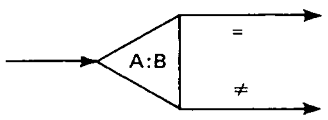
Control statements



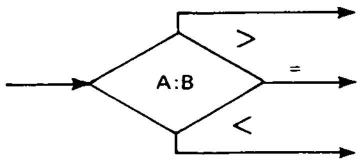
Unconditional transfer: directed arrow to a statement number (sn) enclosed in an octagon

Conditional transfer

The colon is introduced for comparison of expressions followed by directed arrows.

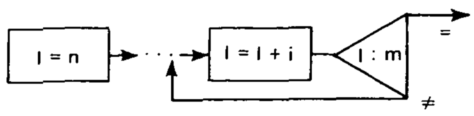


Equality: triangle

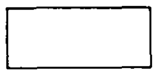


Inequality: diamond

A and B in the diagrams may each be arithmetic expressions. These representations encompass the GO TO and the IF statements.



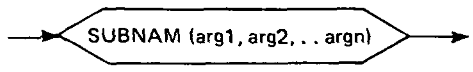
The DO statement is easily represented by a combination of arithmetic and conditional transfer expressions indicating a loop on I from n to m by i.



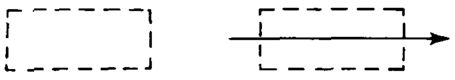
Input/output (I/O) statements: rectangles. Format statements are written and spaced exactly as they are to appear on the printed page, ###.# or .#######; e.g.,

TIME ###.### WEIGHT .######## E##

Declarative statements: no flowchart symbol. These follow the FORTRAN specifications except that the words are prebuilt and the analyst may point to EQUIVALENCE rather than spell it. END indicates completion of the subroutine flowchart.

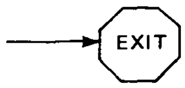


Subroutine execution: hexagon. The CALL or Return Jump is internally generated.



TRANSFER SYMBOL

Comments: perforated rectangles. These statements may be placed anywhere on the flowchart such that they do not interfere with any real statement except the transfer symbol.



Exit: octagon. This is placed on the flowchart where a RETURN statement is to be simulated.

CAPD Graphics

The graphics program automates the flowcharting process by providing space for insertion of new statements and by collapsing the flowchart or extending the arrows as erasures are made.

Today's technology provides at least three methods for sketching and writing on a display device.

(1) Typewriter–alphanumeric characters are immediately available; flowchart symbols could be defined as–

[ ] = rectangle	[= beginning	] = end
<> = diamond	< = beginning	> = end

3 of 3

(2) Menu–characters and symbols are presented on the display and they are initialized by pointing to them with a light pen and indicating a location; e.g., pointing at a rectangle causes a nominal sized rectangle to appear on the display face where indicated, and statements are written inside it.

(3) Character recognition–the analyst draws a rectangle that is "recognized" by two nearly horizontal and two nearly vertical lines; the display presents an internally blocked and sized flowchart symbol or alphanumeric character at the current cursor location.

In either of the latter two methods the flowchart symbol size is modified by "pulling on its handle." A similar method may be devised for moving the symbol with its expressions intact to a different location on the display.

The method of character recognition is probably best suited to the normal analyst. Flowchart symbol recognition combined with typewriter alphanumeric input may be most efficient for keypunch operator use. A menu of flowchart symbols combined with typewriter alphanumerics may seem simplest to implement.

CAPD I/O

In program design it must be possible to initiate a flowchart and at a later date reintroduce it to the computer for additional development.

Inputs to CAPD consist of a graphics flowchart created by the analyst at the console and a previously executed flowchart as output by CAPD. Conceivably, an optical scanner device could be programmed to re-create the graphic flowchart identical to the original; otherwise, an alternate form of input would be made available.

Outputs from CAPD consist of hard copy of the graphic flowchart (optional computer output microfilm); source language for the appropriate compiler (possibly an option of punching the source language on cards); and, in the event that an optical scanner is unfeasible for inputting a previously generated flowchart, a representation of the flowchart on cards, tape, or other media.

The only restriction applicable to I/O is that CAPD generates certain output such that it may become input to itself.

CAPD Diagnostics

A flowchart convention may exist to prohibit the analyst from leaving an unfilled flowchart symbol; thus, if he does not know precisely what the symbol is to contain, he writes a question mark (?) and is allowed to proceed. This permits CAPD to examine each flowchart symbol and report omissions or errors as they occur. Validation of the calling sequence arguments with library subroutines is a potential diagnostic. When CAPD receives the END signal, it questions the analyst concerning the unfilled flowchart symbols, the undefined transfer points, and formats. The diagnostic output provides the analyst with the correct format of any symbol or expression and automatically displays itself if he commits an error repeatedly.

Successful compilation is almost assured; successful execution depends on the response to the diagnostics. The analyst may elect to ignore or leave incomplete portions of the subroutines; CAPD will not inhibit use of the compiler if the user specifically requests to proceed.

Programming Reliability

The diagnostic remarks assist in immediate error recovery. Pictorial representations are considerably less error prone than word images. Modifying a flowchart, where such is possible without copying it, is nearly always performed accurately, whereas generating the code requires particular attention to the format, punctuation, and logical assumptions of the language.

Quick Response and Rapid Reaction

The calendar time required to design or modify a program is drastically reduced by automated regeneration of flowcharts as refinements or alterations are introduced and automated translation of flow diagrams into source language. The implementation of a new or revised set of guidance equations, for example, would take relatively little time compared to today's normal turnaround time. This technology provides real-time systems with fast and accurate response.

Documentation

CAPD reverses the entire documentation procedure. Contracts normally oblige the analyst to document the program. With CAPD the modus operandi is to "program the document." Flowcharting the subroutine is no longer necessary, and the remaining documentation would be formatted as described in ADS to permit complete automation. Thus, the portions of documentation that are always tedious and laborious to produce are bypassed. A very important result is that the readability and reliability of the program are greatly enhanced.

CAPD Recommendations

The implementation of CAPD should be seriously considered by developers of computer software. Not only would the state of the art take a major stride forward, but considerable cost effectiveness would ensue from the diminished time required to create, update, maintain, and document a program.

RECOMMENDATIONS AND CONCLUSIONS

The computing industry has already accrued considerable benefit from the ANSI standards for the FORTRAN language; programs that adhere to these specifications transfer readily to another computer system. There should be similar standardization for loaders. As yet ANSI either has not addressed this problem or has not felt sufficiently fortified to assert itself to this technically feasible but politically delicate problem.

NASA, an important customer of the computing industry, now asks whether program documentation in a broad general sense can be automated. The response is assuredly positive if standardized specifications can be outlined and accepted. Of course, each software developer has and can continue to automate his program documentation individually, but from the buyer's viewpoint cost effectiveness is not necessarily achieved. Certain standardizations are considered in the following recommendations.

Program Specifications

The specifications for programs contained in requests for proposals occasionally present problems to the responder by putting him in doubt as to the relative complexity, generality, and capability of the product desired. This author recommends the topic as the subject of a future symposium.

Manufacturer Software

Much internal program information that is immediately available from the compilers and loaders is lost simply because it is not printed. As discussed in the section entitled "Automated Documentation of Program Internal Communication," forward and reverse subroutine and common reference information is invaluable documentation. It is recommended that the manufacturer of software systems provide options to retrieve the information so that all applicable items in table 1 may be marked affirmatively.

Program Development

The CAPD system discussed in the CAPD proposal should be seriously considered for the best use of engineer and analyst time and to reduce a major portion of the effort expended in documenting a program.

Program Documentation

Many items of subroutine level documentation are considered standard; such as the name, title, author, abstract, calling sequence, restrictions, and variables usage. The format of these information files remains to be defined; this author recommends coded comments.

It may be presumptuous to anticipate that the industry could agree on those items to be retained in the program listing, in the flowchart, and in the documentation. Therefore, the code should contain options that permit the commentary to appear on the respective documents as requested, thus avoiding duplication without sacrificing completeness. The coded comment cards defined for the ADS program discussed represent a step in the correct direction, but additional refinements along with modifications to the current compilers, flowcharters, and documentation programs must be specified, standardized, and implemented.

Manuals Preparation

Input, output, and deck setups are fairly common subject titles in most users manuals. Having directed the development of a large, complex, general-purpose program, the author is aware of some inadequacies of this type of manual. The customer requires problem-oriented information.

It does seem clear that manuals should be generated with automated text editors as mentioned earlier in the paper. An economy of operation is realized, particularly for high-usage dynamically evolving programs, when the documentation can easily and accurately be created and updated.

DISCUSSION

MEMBER OF THE AUDIENCE: I would like to know to what extent the information you have here is available to the general public. In other words, is it proprietary?

LANZANO: Yes, the programs themselves are proprietary. They are for sale.

MEMBER OF THE AUDIENCE: You indicated that this was tied directly to your trajectory determination program. Is there anything within that program that restricts its use?

LANZANO: I did not mean to imply that it is tied directly to this program. It was developed to support this program. It would support any program that follows the set of standards that I defined. We also have other programs that will handle all machine-language programs in a somewhat similar manner; therefore, they are strictly supportive programs.

MEMBER OF THE AUDIENCE: This is quite an involved system. How many years has this been in process?

LANZANO: I think it has actually been in process for 3 or 4 years. They are not difficult programs to write. Most of them have been written as kind of off-the-cuff things. The ADS program is probably the more difficult because it mimics the compiler. In direct answer to your question, it has evolved over a period of years, but the level of effort in producing this is not particularly high.

MEMBER OF THE AUDIENCE: In this area of standardized loads, do you think an extension in the FORTRAN standard that identifies the requirements for overlays and segmentation would be useful?

LANZANO: I would like to see some standards defined for overlays and segmentation. Whether such standards would be useful would depend. I have found the IBM 360 linkage editor to be quite versatile. I think it is probably one of the best in the field right now. The Univac 1108 looks very good to me, although I have not actually used it.

MEMBER OF THE AUDIENCE: The only other question I have is in the area of the technical editor. Have you had any occasion to wish that the technical editor would do a quick index for you on the document, so you could actually go through and take your document and see how well you have used the same phraseology so that it becomes easier for the reader?

LANZANO: This one does not, but there is one called QED developed by Time-Share that will look for phrases.

MEMBER OF THE AUDIENCE: I mean a quick index of the whole document from the form that you put it in.

LANZANO: I believe the one that Dr. Rich discussed actually put out a table of contents at the end.

N73-19221

AUTOMATED ENGINEERING DESIGN (AED): AN APPROACH TO AUTOMATED DOCUMENTATION

Charles W. McClure
SofTech

The automated engineering design (AED) system is essentially a system of computer programs designed for use in building complex software systems, especially those requiring the development of new problem and user-oriented languages. The AED system itself may be considered to be composed of a high-level systems programming language, a series of modular, precoded subroutines, and a set of powerful software machine tools that effectively automate the production and design of new languages. The AED language itself is a modification of ALGOL 60 to make it suitable for use as a systems programming vehicle.

The AED system was developed by Douglas T. Ross and associates over a 10-year period while they formed the nucleus of the Computer Applications Group at the Massachusetts Institute of Technology's Electronic Systems Laboratory. Before their work on AED, this group had developed the automatically programmed tools system for numerical control of machine tools. So, it is natural that people with such extensive experience in the data processing field would be concerned with the documentation of their own activities and with providing facilities for the users of their work to effectively document their own new developments in the software field.

It is necessary to define what is meant by documentation before it is possible to discuss automated documentation. In a general sense, the term documentation refers to everything that can be used by a human being to help understand a computer program or system. This specifically includes flowcharts, listings, cross-references, and, of course, user manuals. This discussion will be limited to the more mechanical aspects of the total documentation spectrum. User manuals, which are more appropriately considered in terms of the psychology of human learning and communication, will be excluded from consideration.

In examining the process of software production, it may conveniently be divided into the design, programming, testing, and production phases. This is illustrated in figure 1. Current manual documentation methods ignore the process of documentation during many of these phases. For an automated documentation facility to work properly, data that are generated during all these phases must be captured and reasonably ordered. This is especially relevant in the programming and test phases, in which numerous individuals are involved and in which errors, oversights, and shortcomings in the original software design are discovered. In addition, the need for new and revised features unfolds as the understanding of the problem matures through usage. These new and potentially valuable insights are frequently lost during the current process, which tends to ignore documentation at these points in the process.

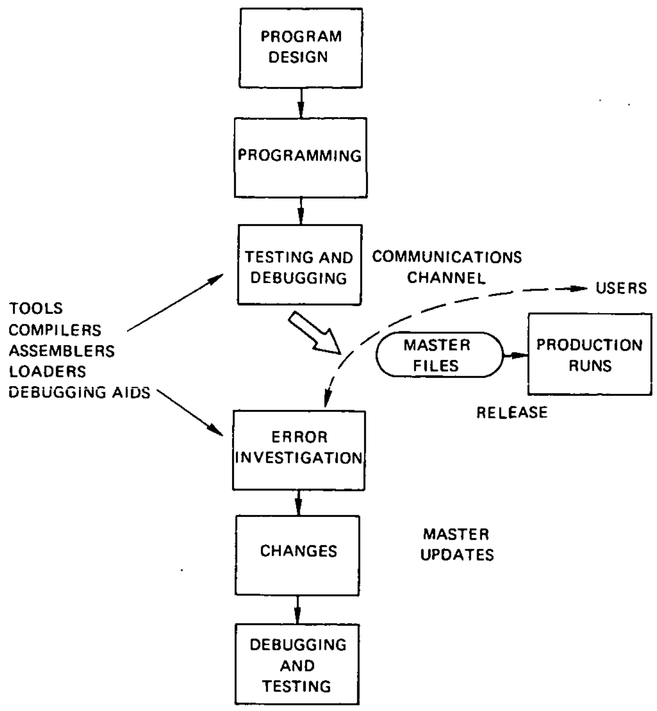


Figure 1.–Software production phases.

SofTech strongly believes that it is erroneous to view the problem of system documentation as an annoying, mundane, and unfortunate concomitant of the software development process. On the contrary, documentation is an important, challenging, and integral part of the total software production picture. Major system software efforts are among the most complex and difficult projects undertaken by man, especially when real-time physical phenomena as well as the vagaries of human behavior play an important part in the overall successful operation of the system. In software, the design-to-production cycle can never be considered completely finished.

Even if present efforts toward the development of highly reliable programming techniques are fully successful in the future, for major systems, the continuing evolution of new user experiences that must be reflected in modified system behavior still will cause the documentation function to be of primary and lasting importance. In SofTech's view, documentation is a natural part of the life cycle of a software system that comes to the fore not at a certain stage of maturity of that system but must be essentially functional as a useful problem-solving tool at each stage.

SofTech's many years of experience in major system development, maintenance, and distribution have given rise to a clear definition of the ultimate goal toward which we strive. This goal is the automated production of high-quality reliable and functional software in a "software factory" environment with the automated production of documentation as an integral process. It is not surprising that the production tools of the documenter and the application of those tools in an orderly technology are closely related to the similar functions required at the generative stages of software design and prototype construction. In fact, it is the purposeful extraction of pertinent information from those production tools as an easily acquired byproduct of the production process that best provides an orderly approach to the solution of documentation problems. This is apparent because the documenters must mentally disassemble a working major software system into its component parts and examine the behavior of those parts in concert as well as separately. At the present time this analysis is supported only by an informal collection of listings, flow diagrams, and sundry debugging aids, most of which have been humanly generated in a sometimes well-directed, but all too frequently ineffective, effort to provide the needed information.

Quite a different state of affairs would prevail if the vast amounts of relevant information which appeared briefly and then evaporated in the process of initial construction and assembly had been saved as a part of the development process. For example, the documentation job would be greatly facilitated if the compiler used to compile a program left in a data bank not only the basic symbol table information but also the cross-reference information concerning external functions defined in or needed by that compilation, the linking loader had left behind its records of program module placement, and the system generation program had retained all relevant information including commentary by the programmer who made some last-minute adjustments, for example, to overlay linkage characteristics. Furthermore, if all of this information were ordered and collected in an automatically generated data bank as a byproduct of the software machinery, it would be essentially free of human error. This information could be combined with other features of the software production methodology, such as the requirement that appropriate commentary be incorporated in each module of a hierarchically assembled system. Before such an assembly operation would be considered complete and acceptable to the system, the body of information available to the system would attain some real substance and would provide a firm basis for a well-structured approach to the documentation task.

The value of this approach can be appreciated by noting that the construction of overlays (to permit a program to execute within a core memory constraint) has been a problem for a great many programmers using various computer systems. Yet an acceptable overlay structure can be produced quite mechanically if the program and data interrelations are clearly stated. This acceptable structure may be improved if frequency of usage information is available.

Programmers attempting to document their activities can benefit by having a high-level language uniquely tailored to order their thoughts and supply direction to the computer. The nouns of such a problem-oriented language are such things as program modules and system assemblies. The adjectives of the language are the various states of those modules, such as unedited source deck texts with commentary. The verbs correspond to the control of the various assembly operations cited above as well as the disassembly operations needed to unfold a structured program to gain access to internal points. With this accomplished, software probes or statistical measuring submodules nested within this language and interlocked with its features can be employed. Once a change has been made, the same set of tools controlled by the high-level language will record the updated information about what was changed, why, and how, as the single change must be reflected throughout the entire system. This language could easily be used through a graphics-oriented terminal, thus providing the combined capabilities of high-level metalanguage with the most effective computer hardware man-machine communications capability available.

This description of the documentation function as an integral and natural part of the software factory of the future may seem utopian at first, but such a view does not do it justice. Instead, it provides the overall vision whereby order can be brought out of our present chaos. It provides a yardstick whereby current and proposed methods can be evaluated and also provides a concrete series of subgoals, many of which are immediately achievable as natural adjuncts to present documentation methods.

Another major aspect of the documentation retrieval problem that also merits particular attention is the language in which the system itself communicates its wealth of information back to the programmer. It is in this area that the various forms of computer graphics can play a particularly important role. The thought processes that accompany the detective work associated with debugging and design can be rapidly overloaded by unending detail. The information must be presented in a way that is most readily assimilated by the human mind. In debugging, even more so than in many other activities, it is very important that the programmer be able to maintain intellectual momentum in pursuit of an idea. This can be greatly aided by use of on-line graphic displays. Here again, the key to successful efforts must lie in the conscious treatment of the graphical displays as a proper language, not merely as a picture-making capability, coupled with appropriate information and data structuring for the data base.

It should thus be obvious that the data base must be content addressable, or be provided with "backward" threaded lists. Perhaps a simple example will demonstrate this need. Consider that the description of a tape file has been stored in the data base so that all programs that are to manipulate that tape obtain the details of its format from the data base. It is clear that reassembling the program will not only incorporate the latest changes to the program but also all modifications to the file description. But how are we to know which programs should be reassembled unless the data base retains the information about which programs do in fact rely on a generic file description?

THE IDEAL ENVIRONMENT

The ideal environment and long-term future goal is a comprehensive, totally computer-automated system that provides a rich, system-independent metalanguage for assembling software systems, tearing them apart, examining them under specific test conditions, modifying master source program files, and other tasks. The nouns in this control metalanguage are master file names, loading and overlay patterns, and test procedures, while the verbs include compiling, loading, updating, debugging, and data base query actions.

The system design provides that any changes made to a master file will be rejected by the system unless all required updates are completed, including documentation updates and updates to related programs affected by the original change. Management control reports are also automatically provided, such as reports of schedule slippage and accuracy of maintenance estimates versus actual performance.

The primary user control device for this system is a graphics terminal, which is particularly adapted to rapid rifling through source files, program editing, display of program flow, and documentation updating. The research in machine-aided cognition by Dr. Engelbart at Stanford University has demonstrated the power and utility of such devices.

The behavioral scientist will play a central role in the development of the language by providing the needed human engineering. He will insure that the language is properly matched to the skill level, human limitations, and working conditions of the maintenance programmers. He will also criticize the management control reports and the user/maintenance project communication facilities to tune these important interfaces for maximum compatibility.

The data base required for effective operation of the maintenance facility is initially constructed by the computer by stripping off program information at each stage of software production, somewhat along the lines of the COMPOOL facility of JOVIAL and the AED insert file control mechanism. Compilers, assemblers, loaders, and other software production tools already produce listings, load maps, subroutine cross-reference printouts, and other aids as natural byproducts of their function. These valuable aids form a major portion of the needed data base, although at present these data are not retained and cross-indexed automatically by the computer for later use. Original program documentation and facts discovered at each stage of the maturing of the software form another important component of the data base.

FACTORS WHICH INHIBIT EFFECTIVE DOCUMENTATION

The engineered documentation environment described above is at present not a reality. Without engineering tools and disciplines, documentation efforts suffer from several inhibiting factors including–

(1) Lack of program understanding by programmers, compounded by poorly engineered and programmed software
(2) Lack of automated software tools for maintaining, updating, and distributing documentation of systems to users
(3) Communications gap among the programmers about status of known errors and correction schedule
(4) Lack of automated testing techniques to insure that modifications do not in themselves create additional errors
(5) Interference and cross-coupling between several programmers simultaneously changing and correcting related programs
(6) Inability of programmers to reproduce error conditions, at least in a sufficiently simple environment to isolate a single problem effectively

THE PATHWAY TO THE GOAL

Given the gap between the ideal environment and the present-day facilities, an evolutionary path to the goal is needed. A graceful degradation of the ideal to fit the present-day hardware and operating system limitations provides a usable facility whose expansion and elaboration insure the needed incremental progress toward the ideal. Many of the programming aids needed to carry out the actions of the metalanguage already exist in present computer software libraries:

(1) Automated flowchart aids such as AUTOFLOW to produce quick, accurate flow diagrams directly from source program statements
(2) Machine-language listings showing the primitive steps compiled to carry out statements in a high-level language
(3) Symbol prints to show memory locations and other values associated with user variables
(4) Load maps showing program layout when executing the program
(5) Cross-reference maps showing interactions among program modules
(6) Library statistics showing the names and sizes of programs called in from software libraries
(7) Traces of program flow showing argument values and machine conditions at selected points during program execution
(8) Linkage tables showing names, locations, and lengths of selected programs
(9) Timers and program counters which illustrate dynamic characteristics of program operation

It is clear that many of the needed software tools are now available. What is missing is the overall system which will permit each of the components of the system to retrieve the data that it needs to perform its function without asking for additional information from the programmers. It is clear that if processors are going to obtain information from the data base, then it is desirable that each processor store information in the data base for the use of other processors. This is the essence of the software factory of the coming decade.

The software factory provides the framework within which the programmer functions and, in particular, should provide both the present processors and the data base discussed above.

SofTech feels that the AED approach permits us to work toward this final goal in the most direct manner now possible.

N73-19222

PROGRAM ANALYSIS FOR DOCUMENTATION

G. H. Lolmaugh
Programming Methods, Inc.

Program analysis for documentation (PAD) is a technique that produces computer program documentation in three steps. It is FORTRAN oriented but could just as well be directed toward any other programming language. It currently gets little help from the computer, but this is hopefully only a temporary hiatus in its development cycle. The three steps to the program analysis include describing the variables, describing the structure, and writing the program specifications. This is clearly the opposite order of the normally accepted way of doing things, but is consistent with the way much programming is done.

The questions of why or how programs get written with or without beforehand understanding of the problem statement or why one should bother documenting the intricacies of something that works will not be discussed here. Only the premises that a program exists in compilable form and a decision has been made to document it will be of importance.

Before proceeding, it should be made clear that a program that contains few or no lucid internal comments, for which no programmer's notes are available, and whose author is unavailable, is going to be difficult, if not impossible, to document properly. But this cannot be used as an excuse for refusing to document such a program. Facts will become known as the calling and called routines are analyzed, and after two or three analysis passes through the entire system are made, much information will emerge. Articulate documents can appear when an organized system is consistently used for posting facts as they are found.

Large systems containing many routines usually have a more or less elaborate scheme of blocked common variables. It is convenient for the documenter to develop a completely separate common document where all the known facts about each common variable are posted. This makes it unnecessary to describe a common variable thoroughly and redundantly in each routine that uses it.

Programmers will undoubtedly balk at the notion of describing every variable. For the moment, it will be assumed that every variable is important, or it would not be in the program. Unimportant and abandoned variables have been known to cause trouble. Being able to decide what to leave out of a document without compromise is what makes a documenter a skilled professional. As this analysis proceeds, a method will be developed for specifying in advance the criterion of impunity.

LISTING THE VARIABLES

The variables in a FORTRAN program have several attributes that are of interest to the documenter. These attributes are well known to the compiler, and, in fact, the compiler produces, or can be directed to produce, listings in various orders of sort and with various degrees of completeness of facts. A proposal for an additional listing will be specified shortly that will save much of the labor and tedium about to be described.

Describing the terms to be used at the beginning of a technical report has been an accepted standard procedure for many years. This enables the author to use his terms as symbols in the body of his report and keep his report concise, orderly, and fast moving. With PAD, a program writeup can be organized in the same way. Describe the terms (variables) first, then write the body of the report (program specifications). This may seem backward to those who view specifications as having been written before the program is written, but it does not seem at all backward to the programmers, users, and other technicians for whom the writeup is, after all, being written.

Each of the variables in a FORTRAN program may be described as either arguments, common, or internal. The first two may be classed together with the read and write statements, as input/output (I/O). I/O, it turns out, is the single class of information of most concern to the greatest number of readers.

DESCRIBING THE STRUCTURE

The structure of a program is simple or complex in proportion to the amount and complexity of the branching being done. This includes looping, which is a special form of branching. If few branches are involved, or the program is short, the description of the program structure need not be separately documented. However, if more than 5 IF's or DO's are present in the program, an author is well advised to construct a flowchart. The automated flowchart programs available, particularly AUTOFLOW, may be used to advantage.

Hand-produced flowcharts, neatly and precisely drawn, lend considerably to the credibility of the finished document, particularly if the flowchart is accurate. The choice of symbols and shapes used is of less importance than consistency and style. Above all, a flowchart should flow.

For very complex programs, particularly those more complex than they should have been, an automatically produced flowchart is probably more economical and more accurate. However, unless some nesting or editing technique was employed in generating the flowchart, it may be difficult to follow. Flowcharts should never be typed, except by an illustrator or technical typist especially trained to do this work.

WRITING THE PROGRAM SPECIFICATIONS

Program specifications should be easy to write after a program is written, but they seldom are. The fact that they were not written beforehand usually means that a program is something less than well designed and orderly. The foregoing descriptions of variables and flowcharts, together with compiled knowledge of called and calling routines, data formats, tabulated output, error messages, and other researched material, make it now possible to outline the problem statement the programmer wishes he had available when he commenced work. How much flesh appears in this outline will depend on how much research he has had time to do, how thorough was his work, and how many times he has reedited the writeup for each interrelated subroutine in the system.

Because this paper deals with program analysis, it will not endeavor to show how to do technical writing. At this point, the program analyst must decide who his readers are. Appendix A describes several possible readers and some of their needs. A good checklist of topics and objectives should include the following:

(1) A statement of the mathematical model, equations, and formulas. If copied from, or based on, a textbook case, copy the material here, with credit given in the references.
(2) A statement of the technique, such as sorting or merging, and a description of the sort key or collating sequence.
(3) The decision criteria and tables.
(4) A description of the broad aspects of the I/O consistent with the details already specified.
(5) A stressing of what the program does, rather than how it is done, except where the means of accomplishment is tricky and will not be immediately obvious by examining the code listing.

MANUAL AND AUTOMATED METHODS

The current manual PAD method and the preliminary specifications for a proposed compiler option are described in appendixes B and C, respectively.

APPENDIX A—TECHNICAL EDITOR'S NOTES ON EDITING CRITERIA FOR REVIEWING PROGRAM DOCUMENTATION

Review the writeup on the basis of the needs of the intended reader(s). The possible readers and their needs include:

(1) *The maintenance programmer*—adds, deletes, and changes the program on the basis of new specifications. He needs to know, in addition to what the program does and how it works, the impact of any change he may make on other programs. Any I/O variable (argument, common, read/write) may affect any calling or called routine. Changes in logical tests may change the meaning of messages.
(2) *The user programmer*—needs calling sequence details, as well as other interfaces required to transplant a routine to another system.
(3) *The reprogrammer*—will be involved in transplanting this system to a next generation computer, probably rewriting portions of the code for optimization or new specifications. Needs considerable information about the current program specifications.

(4) *The system user*—may require additional information not found in the user's manual to pinpoint unusual data trouble, machine trouble, compiler trouble, etc.

(5) *The mathematician-analyst*—needs to know whether this program does exactly what he wishes to do, avoids what he wishes to avoid, is otherwise suitable to his needs, or holds promise of being suitably modifiable.

(6) *The project director, technical writer, and others*—responsible for writing and maintaining user's manuals, maintenance manuals, and other technical reports. The programmer's workbook should be the major repository of information and should be sufficiently up to date to enable the compiling of reports on short notice.

APPENDIX B—TECHNICAL WRITER'S NOTES FOR PAD

To methodically analyze (document) a FORTRAN subroutine:

(1) Compile the code. A source listing and a cross-reference are needed.

(2) Make a Xerox copy of the source listing. The ISN's and the statements will fit the 8½-in. width of standard paper. Omit card numbers.

(3) Get red, green, and black thin line marking pens.

(4) Underscore the Xerox copy of the listing in green for common block names, calls, returns, entry points, and other program interface elements; in red for read, write, and format statements, name lists, and other hardware interface elements; and in black for IF's. Bracket the DO loops in black.

(5) Build an argument-list skeleton:
   - (a) List each variable in order of its appearance in the calling sequence.
   - (b) Note any word size or mode other than implicit.
   - (c) Show dimension. If equivalenced, consider so noting.
   - (d) Determine I/O status. Use the rules at the end of these notes.
   - (e) For indicators or flags, assign a three- or four-word plain text descriptive name, then tabulate all values and their meanings. For logicals, only the "usual" condition needs describing, unless the opposite status has other than the opposite meaning.

(6) Build a common-table skeleton. For each variable,
   - (a) Check the name to see that it is actually used by this program. Use the symbol table to see that an ISN greater than the first executable statement is present. Skip unused names.
   - (b) Note any word size or mode other than implicit.
   - (c) Show dimension. If equivalenced, consider so noting.
   - (d) Determine I/O status.
   - (e) Refer to the common writeup for this block; determine that the description here is consistent with that in the common writeup. If the usage here adds to the knowledge in the common writeup, update the common writeup.
   - (f) Note any pertinent comments in this program listing.

(g) Check the usage of this variable in several places in the code (use the symbol table) to see that information in (6(d)) and (6(e)) makes sense and is current.

(h) Describe the purpose or usage, as pertinent to this program. Use a brief, concise, terse form. The common writeup should contain the total information about a variable, and lengthy details should be documented as program specifications.

(7) Build a read/write table:

(a) Make one entry for each possible read or write statement.

(b) For a card, mention what the card is for, how many, any preconditions (IFs), and make reference to the card layout figure.

(c) For printed messages, state the conditions for the message, state the message precisely, and mention or show any tabulations that follow. Obtain sample printouts whenever possible.

(d) For a tape read or write, use the same general rule as for a card read, specifying the record and file structure and referencing a tape layout figure.

(e) For disk, data cell, and similar data sets, explain the define file parameters.

(8) Start building an internal variable table. Add to it as analysis proceeds. Unimportant variables may be omitted. An unimportant variable is one whose purpose or usage is immediately obvious. The use of (9) as the index in an unnested DO loop is obvious.

(9) Construct a flowchart, if necessary. This may not be needed if the program contains less than four decision statements.

(10) Write the program specifications.

(11) Write the error procedures, if any. Clues to error procedures are error messages, flags in the argument list or in common, and STOP and PAUSE statements.

(12) Name the routines calling this one (if known).

(13) Name the routines called by this one. Look for FUNCTION names, note program names for alternate entry points.

(14) Add the following sections, as required:

(a) References

(b) Flowcharts

(c) Attachments including tables, card and tape layouts, and sample output, input, job control language.

I/O rules. When describing a variable in a FORTRAN routine as input and/or output, the following rules apply:

(1) Input and/or output pertains to input or output usage of a variable as seen from this routine's viewpoint.

(2) A variable is input if this routine needs it for computation, testing, or other internal purposes.

(3) If a variable first appears in this routine on the right side of an equal sign, the variable is input.

(4) If a variable first appears in an IF argument, the variable is input.

(5) A variable is output from a routine if the routine changes its value in any way.

(6) If a variable appears anywhere in this routine on the left of an equal sign, this variable is output.
(7) A variable may be both input and output.
(8) An argument is input and/or output in this routine consistent with its usage as an argument in a called subroutine.
(9) Arguments or common variables that appear in read statements are output from the routine because values are (or may be) changed. Arguments or common variables that appear in write statements are input to a routine because a value is expected of them.
(10) I/O is applicable to arguments, common, and, conceivably, registers.
(11) Every routine must have at least one input or output item, or the writeup must explain the discrepancy.

APPENDIX C—FORTRAN PROGRAM ANALYZER FOR DOCUMENTATION (PRELIMINARY SPECIFICATIONS)

The purpose of this program is to produce a checklist of items that must be contained in the program documentation and to include as many facts about these items as may be available. Provision is made for communication between internal and external documentation.

Using information available from the FORTRAN compiler, produce tabulated lists of variables and lists of other items to be covered in the documentation. Lists of variables include calling sequence arguments, common variables, variables in read/write statements, and important internal variables. Other items include text of write statements and comments concerning error procedures.

Calling Sequence Arguments

Generate a list of the argument names showing I/O context, dimension, mode (other than implicit), equivalence, etc., in the following form:

```
I      ARG1                       (E) I*4
O      ARG2(n)                        L*1
I/O    IARG(n)                    (E)
          .
          .
          .
```

where (*n*) is the dimension (if any), (E) denotes some equivalence, and I*4 and L*1 are modes that are not implicit.

Elements of this table are to be printed one per line (double-spaced option), suitable for later manual entry (by the programmer) of descriptive text. When operating in the internal documentation mode, the program will scan the first word of each existing comment card containing a delimiter (–) and include the text of that comment. Variables are listed in order of their appearance in the argument list.

I/O context for each variable is to be labeled I, O, or I/O according to the rules in appendix B. These rules are designed to show a user programmer which values he is expected to furnish and which values will (or may) be changed by this routine.

Common Variables

Generate a list of common variable names showing I/O context, block, dimension, mode (other than implicit), equivalence, etc., in the following form:

```
I     BLOKA   VAR1(n)
O             ARRY(i,j)                 R*4
I/O   BLOKB   VAR                  (E)
I             A
O             C
        .
        .
        .
```

where the meaning of the symbology is now obvious.

The same characteristics and rules as for calling sequence arguments apply here. However, only those variables appearing in an executable statement or equivalenced to a variable in an executable statement are normally listed. On option, list all the variables, showing NR as applicable, to facilitate the building or checking of the complete common directory. Note that the NR test used by the FORTRAN compiler differs because of the executability specification. A variable must appear somewhere in the program following the beginning of the first executable statement to be considered executable here. A format statement is considered nonexecutable, but the variables appearing within it are executable, nonetheless.

Block and variable names are listed in the same order as their appearance in their definition statements at the beginning of the program.

Read/Write Statements

Generate a list of read and write statements naming the data set, format or define file number, etc., in the following form:

```
(27)  W  6   END DATA HANDLER – ELAPSED TIME . . (T) . . . SEC
(12)  R  5   REFDAY(3)
             IPRT                                           L*1
               T =
               F =
```

These entries will mostly provide blank space for the author to write his descriptions. However, the FORTRAN compiler can recognize many elements and post them accordingly.

Messages will be printed verbatim, followed by two blank lines to be used to describe the conditions under which the message is printed. A message is any H-type statement.

Name lists, either in or out, are displayed in the following form:

```
&NAME.

    ARG1
    ARG2(n)
      .
      .
      .
```

with dimension, mode, equivalence, etc., as specified for calling sequence arguments.

Read or write statements that can be recognized as cards will have the variables tabulated in a manner conducive to generating card layouts.

BCD write statements to other than card or printer data sets will have the variables tabulated in a manner conducive to generating tape record layouts.

Direct access data-set references will have the define file statement tabulated in an appropriate manner.

Sample output is usually difficult to find for documentation purposes and often lacks the generality the author wishes because of the conditions of the run. Making a test run in which all cases and all error messages are displayed is challenging. Therefore, in the DOCEXEC mode, all output statements will be executed once, consistent with the included GO and DD statements. Variable quantities in this display will be dots, indicating the field size specified by the format statement.

Switches

All logicals and all I*1 and I*2 variables will have extra line spaces for entering logical conditions in addition to the name of the variable. Logicals will provide T/F indication lines; integers will provide an arbitrary three extra lines.

Arrays

An arbitrary three extra lines will be provided following each subscripted variable name for the purpose of describing the individual variables in an R-type or I*4 (or larger) array.

Care should be exercised here not to generate too much blank paper. For instance, each of the elements of a transformation matrix need not be described. However, in the case of an array, SPEC(20, 7), where the I's are characteristics and the J's are stations, 21 lines should be generated; 1 on which to describe SPEC, and 20 on which to describe each of the characteristics. Conversely, if the I's were stations and the J's the characteristics, eight lines should be generated.

Internal Variables

Generate a list of all internal variables showing dimension, mode, and equivalence. All variables whose names contain two to five characters and that cannot be defined as arguments, common, or read/write variables are considered internal variables. The size restriction permits a programmer to assign variables whose usage will be intuitively obvious without their being forced into the documentation. A PAD calling argument (OLD) will defeat this test for documenting preexisting programs.

Summary

These preliminary specifications are being revised as time, inclination, and additional interest are shown. Readers wishing to participate in developing these specifications are invited to send in their contributions.

DISCUSSION

MEMBER OF THE AUDIENCE: Where do you stand at the moment on the development of this system?

LOLMAUGH: It is an idea.

MEMBER OF THE AUDIENCE: Have you done any development on it?

LOLMAUGH: The manual portion of it (apps. A through C) does exist, and I already use it. I also use it as a tutorial for the tech writers, programmers, or anyone else who helps me with documentation. I had planned to discuss that at greater length because I know the programming people in this group would have liked to hear more about it, but I was discouraged from that because it does not involve automation and this was after all a symposium on automated documentation. I do use a symbol table from compiled routines. I hope my paper was able to illustrate the volume of hand work that I do that I know the computer can help me do. I join several of the previous speakers in requesting that the compiler be put to more work. I think it can be put to more useful work, at least where people want documentation.

N73-19223

TREE-STRUCTURED INFORMATION FILE AND ITS SUBPROGRAM SUBTREE

Charles K. Mesztenyi
University of Maryland

The goal of automatic documentation of computer programs is to establish procedures, called documentation programs, that can be implemented by computer programs. These documentation programs may be divided into two categories: postmortem and developmental documentation programs. In the former case, a computer program is presented as input for documentation without any preparation; in the latter case, the program to be documented must be developed so that it contains information necessary for the documentation.

This paper is concerned only with the development documentation programs. A document tree is defined as the syntactic representation of a document when it is divided into subdivisions such as chapters and sections. A developmental tree is defined as a tree of information obtained during the course of the development of a computer program. The task of documenting a computer program is then made equivalent to a transformation of its developmental tree into a document tree. When this transformation is performed by a computer program, the documentation can be achieved automatically.

There is no attempt made in this paper to define the document tree more precisely. Only its tree structure is assumed. Efforts are concentrated on the developmental tree, specifically a subtree of it; the subprogram tree is illustrated in more detail.

GENERAL APPROACH

In the development of documentation programs, two objectives are paramount. Pieces of information about the program to be documented should be kept in a computer file during the development of the program, and this information should not be duplicated in the file. The importance of the first objective is obvious; the information should be in a computer-readable form for documentation. The importance of the second objective can be seen whenever a change is made during or after the development of the program to be documented. One can easily make the mistake of changing information in one place and forgetting about it in the other place. On the other hand, a change of information at a certain place may require changes in other information.

The goal of this project is to structure the developmental file of information in a tree structure (fig. 1) so that the nodes represent pieces of information. Any change in the

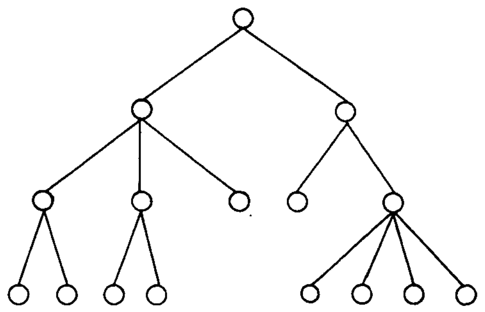


Figure 1.–Tree structure.

contents of a node may require changes in the subtree rooted in that node. In certain cases when the semantic structure is more complex, i.e., it may represent a directed graph, pointers may be used semantically.

The final documentation of a program is produced from its developmental tree of information. A special tree-traversing program, possibly interactive, selects out the contents of nodes or subtrees, invokes certain documentation programs to transform these data into special format, and stacks this information sequentially. The sequentially stacked information is processed by a listing program to produce the final printed document.

Obviously the main problem is the establishment of the developmental tree structure. At this time, a complete tree structure cannot be proposed. The definition of certain types of subtrees, however, has been accomplished. One of these, a source program subtree, is described in detail.

FLOWCHARTING AND PROGRAM LISTINGS

Any large computer program should be segmented into subprograms, subroutines, and procedures. The size of a subprogram may depend on its complexity and on its source language. Documentation of a subprogram is usually done in three different forms: textual description, flowchart, and source language listing.

The information should be structured as a tree. A source program is compiled (assembled), which generates a relocatable program. Figure 2 then defines the tree.

Certain information such as size, entry points, and external references can be obtained from the compiler-generated relocatable program. The rest of the information should be put into the source program. Textual information can easily be placed into the source program by grouped comment lines. Thus the source program may be defined as a tree, as seen in figure 3.

To combine the flowchart with the source program creates some problems. A special

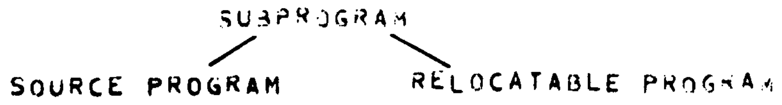


Figure 2.–Tree structure for subprogram.

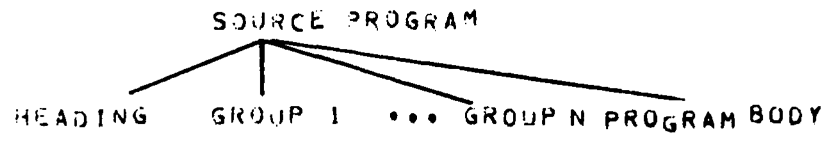


Figure 3.–Tree structure for source program.

form called a sequence chart is used. This is not a complete flowchart in the standard sense, but it forces a tree on the otherwise graph-structured flowchart. Then there is no problem in listing a tree structure sequentially. The missing links of the graph structure, which appear as transfer statements in the source program, can be implemented by semantic comments. A special computer program for a source language can automatically flag these places.

Appendixes A, B, and C show the final printed forms of three different subprograms. The right side of the lists contains the actual program statements; the left side is stored internally as coded comments. The listing program takes care of this separation, but the actual sequential form is kept in the vertical direction. Those flow lines that represent the spanning tree of the program are shown with special characters, colons, periods, and asterisks. The groups of textual descriptions are separated by horizontal lines of asterisks. Both the names of the groups and the characters used for line drawing are made flexible by changing an internal table in the printing program. Special print programs are available: A "level" print gives only those lines that are not indented more than a certain input parameter. A "selective" print gives only a subtree; i.e., a defined group or a subtree of the body. The output of these print routines, formatted for a document processor, can be kept in the computer.

This form of documentation has been very helpful in the project from which these three examples were taken. During the debugging stage, it was easy to follow the sequence chart to locate a specific segment of a subprogram without turning pages back and forth.

Obviously, to get these forms, a good editing program capable of performing insertions and changes is needed. Appendixes D and E show appendix A in a developmental stage. In appendix D the initial sequence chart is defined. In appendix E an update procedure is shown. First the sequence chart is shown in a coding sheet geometrically; then its code is placed in front of it. The code for a line is composed by two fields. The first field defines either the depth of the text, 0 to 9, and blanks for program statements or contains special instructions, like group heading, change, and insert commands. The second field contains subcodes, such as line drawing codes for sequence charts and line numbers for updating commands. The text appears in the third field. In the actual input, the text field gets left adjusted. The lines will not be represented because they are already defined by codes.

This procedure for writing a program has the following advantages:

(1) It provides an up-to-date documentation of the program in the developmental stage.
(2) It forces a programmer to lay out his program so that it provides an automatic documentation at any level.
(3) It provides a form for a project leader to define subprograms without details that can be inserted by other programmers.
(4) It may be used for the present-day coded flowcharting programs.

Its main disadvantage is that it needs more work and discipline in the beginning.

SUMMARY

Printed documents have syntactic tree structures, such as titles, chapters, and sections. The semantic contents of the document may have more complex graph structures, but these structures are implemented by semantic references. A computer program has a graph structure also, but a spanning tree on this graph can be defined with semantic references to the missing links. This developmental tree of a program may have a different arrangement from a document tree. If the necessary information is contained in the developmental tree for the document tree, a transformation program can produce a document tree from the developmental tree. If the structures of the two trees are standardized, then this transformation can be achieved automatically. Otherwise, an interactive transformation routine can achieve a semiautomatic documentation.

APPENDIX A—PRINTED SUBPROGRAM: EXAMPLE 1

```
                                                                SUBROUTINE EXPRES (*, ISW)
******************************************************
             TITLE
EXPRESSION TRANSLATOR, INFIX TO PREFIX
******************************************************
             ABSTRACT
*** AUTHOR: C.K.MESZTENYI
*** DATE: JULY 21, 1970
*** LAGUAGE: FORTRAN 5
*** PROJECT: FORMAL - SUBROUTINE
*** SEARCH KEYS: NONE
******************************************************
             DATA STRUCTURE
FORMAL.CMMN
                                                                INCLUDE CMMN
FORMAL.PWORD
                                                                INCLUDE PWORD
*** ARGUMENT:     *   ERROR RETURN
                ISW   INPUT ARGUMENT
******************************************************
             SPECIFICATION
THIS IS A GENERALIZED EXPRESSION TRANSLATION ROUTINE FROM
INFIX TO PREFIX FORM. IT ASSUMES THAT THE CALLING ROUTINE
INITIALIZED THE SCANNER, THUS GSCANR GIVE THE CONSECUTIVE
LOGICAL SYMBOLS. THE ROUTINE MAY BE CALLED FROM 4 DIFFERENT
PLACES DEPENDING ON ISW:
ISW = 0 PROCESS AN ASSIGN STATEMENT: VARIABLE = EXPRESSION ;
    = 1 TRANSLATE THE EXPRESSION PART FROM A READ-IN DATA
        WHICH MAY BE IN THE FORM:  EXPRESSION ;
                            OR     VARIABLE = EXPRESSION ;
    = 2 PROCESS SUBSCRIPT EXPRESSION IN THE FORM:
                      EXPRESSION )
    = 3 PROCESS AN EXPRESSION IN THE FORM:
                 EXPRESSION ;
IN THE FIRST CASE, THE INFORMATIONS FOR THE VARIABLE ARE
STORED IN N1,N2,N3. IN THE SECOND CASE, ONLY THE EXPRESSION
PART IS RETAINED UPON RETURN. IN ALL CASES, THE TRANSLATED
AND SIMPLIFIED EXPRESSION IS PLACED ABOVE THE PUSH-DOWN
STACK WITH THE PUSH-DOWN STACK CONTAINING ONLY ONE ENTRY:
A COMMA WITH A COUNT CORRESPONDING THE NUMBER OF
EXPRESSIONS TO ACCOMODATE LISTS.
******************************************************
             METHOD
AFTER INITIALIZATION, THE LOGICAL BCD SYMBOLS ARE OBTAINED
BY GSCANR AND PROCESSED ONE-BY-ONE IN A LOOP. PROCESSING A
SYMBOL IS DONE AS FOLLOWS:
   FIRST, IT IS CHECKED IF THE SYMBOL IS IN CORRECT TEXT;
   THEN
CONSTANTS- ARE LINKED IN ABOVE THE PUSH-DOWN STACK;
VARIABLES - THEIR VALUES ARE OBTAINED FROM THE SYMBOL
          TABLE AND LINKED ABOVE THE PUSH-DOWN STACK.
          IF THE VARIABLE IS SUBSCRIPTED, OR IT IS A
          FUNCTION IDENTIFIER, THEN THE NAME IS LINKED
          IN ABOVE THE PUSH-DOWN STACK, AND A LEFT
          PARENTH. IS PLACED IN THE PUSH-DOWN STACK WITH
          COUNT=1.
```

```
LEFT PARENTH. - IS PLACED IN THE PUSH-DOWN
          STACK WITH COUNT=0.
OPERATORS - THE PUSH-DOWN STACK IS EMPTIED OUT BY STKOUT
          UNTIL ITS TOP ELEMENT HAS PRECEDENCE NUMBER
          EQUAL TO OR LESS THAN THE PRECEDENCE NUMBER
          OF THE OPERATOR. THEN THE OPERATOR IS PLACED
          IN THE PUSH-DOWN STACK. SIMPLIFICATION IS
          PERFORMED BY STKOUT.
RIGHT PARENTH., RIGH BRACKET - THE PUSH-DOWN STACK IS
          EMPTIED OUT BY STKOUT UNTIL THE MATCHING LEFT
          PARENTH. IS FOUND. IF THAT HAS A COUNT=0,
          IT IS DISCARDED TOGETHER WITH THE RIGHT
          PARENTH. IF IT HAS A NON-ZERO COUNT, THEN IT
          INDICATES AN END OF SUBSCRIPTS (PAR.) OR END OF
          FUNCTION ARGUMENTS (BRACKET). IN CASE OF END OF
          SUBSCRIPTS, THE SUBSCRIPTS ARE COLLECTED AND
          THE VALUE OF THE SUBSCRIPTED VARIABLE IS
          OBTAINED FROM THE SYMBOL TABLE, WHICH IS
          LINKED IN. IN CASE OF END OF ARGUMENT LIST,
          THE FUNCTION IDENTIFIER IS OBTAINED AND LINKED
          IN
SEMICOLON - INDICATES THE END OF EXPRESSION. THE PUSH-DOWN
          STACK IS EMPTIED OUT BY STKOUT.

******************************************************
             LOCAL VARIABLES


LOGICAL VARIABLE 'SB' IS TRUE WHENEVER THE SCANNED SYMBOL IS
IN SUBSCRIPT LEVEL. 'SBC' VARIABLE CONTAINS THE DEPTH OF THIS
LEVEL.
LOGICAL VARIABLE 'EQL' IS TRUE WHEN AN '=' HAD BEEN PROCESSED
ALREADY, THUS IT MAY NOT APPEAR AGAIN. '=' MAY ALSO NOT APPEAR
ON SUBSCRIPT LEVEL.
THE SYNTAX OF EXPRESSIONS IS CHECKED AT EVERY SCANNED SYMBOL BY
MASKING 'TEST' WHICH WAS SET BY THE PREVIOUS SYMBOL. IF THE RESULT
IS  NOT  ZERO  THEN THE EXPRESSION HAS SYNTACTIC ERROR. IN THE
FOLLOWING TABLE,'A' DENOTES AN ALPHANUMERIC NAME, 'N' DENOTES A
NUMERIC CONSTANT,'I' DENOTES POSITIVE INTEGER:
```

SYMBOL	MASKING BITS (DEC.)		RESET TEST (DEC.)	
INITIAL ASSIGN	---		1000000	(64)
INITIAL OTHERS	---		0100000	(32)
A	0001110	(14)	0001000	(8)
A(	0001110	(14)	0100000	(32)
A[	1001110	(78)	0100000	(32)
N	1001110	(78)	0000100	(4)
[I]	1001110	(78)	0000100	(4)
#I	1001110	(78)	0000100	(4)
(	1001110	(78)	0100000	(32)
=	1110101	(117)	0000001	(1)
UNARY +-	1011110	(94)	0010000	(16)
BINARY +-	1110001	(113)	0010000	(16)
* / **	1110001	(113)	0010000	(16)
,	1110001	(113)	0100000	(32)
) AS SEPARATOR	1110001	(113)	0000100	(4)
]	1110001	(113)	0000100	(4)
) AS END OF SUBS.	1110001	(113)	0000010	(2)

```
             1110001 (113)         ---
    'BRT' AND 'PAR' ARE USED TO COUNT THE BRACKETS AND
    PARENTHESIS, RESPECTIVELY.
    LOGICAL 'NEG' IS SET TO TRUE BY '-' FOR THE NEXT
    CHARACTER SCANNED ONLY.
                                                                LOGICAL SB,EQL,NEG

*******************************************************
             SEQUENCE CHART

INITIALIZE
     PUSH-DOWN STACK WITH COMMA
          :                                                     NP=IGETF1($990)
          :                                                     NPO=NP
          :                                                     C(NP)=20K10
          :                                                     D(NP)=1
     SUBSCRIPT LEVEL
          :                                                     SB= .FALSE.
          :                                                     BRT = 0
          :                                                     PAR = 0
          :                                                     SBC=0
     LOGICAL VARIABLES EQL AND END, INITIAL TEST
          :                                                     EQL= ISW .GE. 2
          :                                                     TEST=32
          :                                                     IF (ISW .EQ. 0) TEST=64
          :                                                     NEG= .FALSE.
     GO TO SUBSCRIPT START IF ISW=2
          :                                                     IF (ISW .EQ. 2) GO TO 180
LOOP TO PROCESS CONSECUTIVE SYMBOL
*    GET SYMBOL
*         :                                               30    CONTINUE
*         :                                                     CALL GSCANR($990,IND,N1,ITC,ICC)
*    BRANCH BY TYPE OF SYMBOL
*    :    IND = 1,2,3,4 FOR
*    :    INTEGER, REAL, IDENTIFIER, SPECIAL CHARACTER
*    :       :                                                  GO TO (100,110,40,60),IND
*    :... INTEGER
*    :      :                                             100   I=0
*    :      :                                                   GO TO 120
*    :... REAL
*    :      :                                             110   I=3
*    :    LINK IN CONSTANT
*    :      :  :                                          120   IF (AND(TEST,78) .NE. 0) CALL FMLERR($990,N1,I,1)
*    :      :  :                                                TEST=4
*    :      :  :                                                J=ILINK1(NP,I,N1)
*    :    CHANGE SIGN IF NEG IS TRUE
*    :      :  :                                                IF (NEG) D(J)=-D(J)
*    :      :  :                                                NEG= .FALSE.
*    :      :  :                                                GO TO 30
*    :... IDENTIFIER
*    :    CHECK IS -1 FACTOR SHOULD BE LINKED IN
*    :      :  :                                          40    INEG = 1
*    :      :  :                                                GO TO 500
*    :    ERROR IF IT HAS MORE THAN 6 CHARACTERS
*    :      :  :                                          50    IF (ICC .NE. 0) CALL FMLERR($990,N1,1,2)
*    :    BRANCH BY TERMINATING CHARACTER
*    :    :                                                     GO TO (130, 180, 190), ITC + 1
*    :    :... IDENTIFIER NOT TERMINATED BY ( OR [
*    :    :         :                                     130   IF (AND(TEST,14) .NE. 0) CALL FMLERR($990,N1,1,1)
*    :    :         :                                           TEST=8
*    :    :         :                                           N2=0
*    :    :    CHECK IF ITS VALUE MUST BE LINKED IN
*    :    :    :                                          140   IF (EQL .OR. SB) GO TO 160
*    :    :    :... NO, GET ITS NAME AS VALUE
*    :    :    :         :                                150   IF (N2 .NE. 0) CALL ILINK1(NP,N2+7,N3)
*    :    :    :         :                                      J=6
*    :    :    :         :                                      IF (N2 .NE. 0) J=7
*    :    :    :         :                                      CALL ILINK1(NP,J,N1)
*    :    :    :         :                                      GO TO 30
*    :    :    :... YES, GET VALUE FROM SYMBOL TABLE
*    :    :         IF UNASSIGNED, THEN GET
*    :    :         ITS NAME AS ITS VALUE
*    :    :              :                                160   CALL SYMBOL($990,1)
*    :    :              :                                      IF (EPTR .EQ. 0) GO TO 150
*    :    :         COPY EXPRESSION AND
*    :    :         LINK IT WITHOUT LEADING COMMA
*    :    :              :                                      II=ICOPY0($990,EPTR)
*    :    :              :                                      I=NEXT(II)
*    :    :              :                                      J=LASTXX($990,II,1,0)
*    :    :         IS IT A LIST
*    :    :         :                                           IF (H2(II) .EQ. 1) GO TO 170
*    :    :         :... YES, EMPTY PUSH-DOWN STACK
*    :    :         :    COMBINE COUNT FOR COMMA
*    :    :         :         :                                 CALL STKOUT($990,18)
*    :    :         :         :                                 IF (ITYP(NP) .GT. 17) CALL FMLERR($990,N1,1,1)
*    :    :         :         :                                 D(NP)=D(NP)+H2(II)-1
*    :    :         LINK IN EXPRESSION
*    :    :         :    :                                170   CALL RMOVF1(II)
*    :    :         :    :                                      CALL ILINKN(NP,I,J)
*    :    :         :    :                                      GO TO 30
*    :    :... IDENTIFIER TERMINATED BY LEFT
*    :    :    PARENTHESIS :   A(
*    :    :         :                                     180   IF (AND(TEST,14) .NE. 0) CALL FMLERR($990,N1,1,1)
*    :    :         :                                           PAR = PAR+1
*    :    :         :                                           TEST=32
*    :    :    SUBSCRIPTED VARIABLE, LINK IN NAME
*    :    :    AND PLACE '(' WITH COUNT 1 INTO THE
*    :    :    STACK, INCREASE SUBSCRIPT LEVEL
*    :    :         :                                           NP=ILINK1(NP,17,1)
*    :    :         :                                           CALL ILINK1(NP,7,N1)
*    :    :         :                                           SB= .TRUE.
*    :    :         :                                           SBC=SBC+1
*    :    :         :                                           GO TO 30
*    :    :--- IDENTIFIER TERMENATING WITH LEFT
*    :         BRACKET :   A[
*    :              :                                     190   IF (AND(TEST,78) .NE. 0) CALL FMLERR($990,N1,1,1)
*    :         GET FUNCTION IDENTIFIER,
*    :         BRANCH BY TYPE
*    :         :                                                I=IFUNCT(N1)
*    :         :                                                IF (I .EQ. 0) GO TO 210
*    :         :                                                BRT = BRT+1
*    :         :                                                IF (I .GT. 1) GO TO 200
*    :         :... DIFFERENTIAL FUNCTION
*    :         :         :                                      NP=ILINK1(NP,23,0)
*    :         :         :                                      GO TO 240
*    :         :... MATH. OR FORMAL FUNCTION
*    :         :         :                                200   NP=ILINK1(NP,21,0)
*    :         :         :                                      H1(NP)=I
*    :         :         :                                      GO TO 240
*    :         :... NONE OF ABOVE, CHECK SYMBOL TABLE
*    :              :                                     210   N2=0
*    :              :                                     220   CALL SYMBOL($990,1)
*    :              :                                           BRT = BRT+1
*    :              :                                           IF (EPTR .EQ. 0) GO TO 230
*    :              :... DEFINED FUNCTION
*    :              :    LINK IN EXPRESSION
*    :              :         :                                 II=ICOPY0($990,EPTR)
*    :              :         :                                 I=NEXT(II)
*    :              :         :                                 J=LASTXX($990,II,1,0)
*    :              :         :                                 NP=ILINK1(NP,22,D(II))
*    :              :         :                                 CALL ILINKN(NP,I,J)
*    :              :         :                                 CALL IFREE1(II)
*    :              :         :                                 GO TO 240
*    :              :... UNDEFINED FUNCTION
*    :                        :                           230   NP=ILINK1(NP,24,N1)
*    :                   LINK IN COMMA FOR
*    :                   THE ARGUMENTS FOLLOWING
*    :                        :                           240   NP=ILINK1(NP,16,1)
*    :                        :                                 TEST=32
*    :                        :                                 GO TO 30
*    :... SPECIAL CHARACTERS
*         BRANCH BY THE SPECIAL CHARACTERS
*         :                                               60    GO TO (270,280,290,270,300,340,350,270,440,270,270,270,390,400,
*         :                                                    1   420,270,270,270,270,430,270,460,460,410,270,270,270),N1
*         :... ILLEGAL SPECIAL CHARACTERS
*         :         :                                     270   CALL FMLERR($990,N1,1,1)
*         :... LEFT BRACKET OR ID. ENCLOSED IN BRACKETS
*         :      [   [I]
*         :         :                                     280   IF (AND(TEST,78) .NE. 0) CALL FMLERR($990,N1,1,1)
*         :    UNDEFINED FUNCTION ARGUMENT OR SUBSCRIPTED
*         :    FUNCTION CHECK IF IT IS SUBSCRIPTED
*         :    FUNCTION
*         :    :... YES, GO TO FUNCTION PART
*         :    :    OF DEFINITION
*         :    :         :                                      IF (TEST .EQ. 2) GO TO 220
*         :    :... NO, IT IS A DUMMY ARGUMENT,
*         :         THEN IT MUST BE FOLLOWED BY
*         :         AN INTEGER AND RIGHT BRACKET
*         :              :                                      INEG = 2
*         :         CHECK IF -1 FACTOR IS NEEDED
*         :                                                     GO TO 500
*         :              :                                285   CALL GSCANR($990,IND,IDT,ITC,ICC)
*         :              :                                      IF (IND .NE. 1 .OR. IDT .LE. 0) CALL FMLERR($990,ITC,1,1)
*         :              :                                      I=ILINK1(NP,5,IDT)
*         :              :                                      CALL GSCANR($990,IND,IDT,ITC,ICC)
*         :              :                                      IF (IND .NE. 4 .OR. IDT .NE. 3) CALL FMLERR($990,ITC,1,1)
*         :              :                                      TEST=4
*         :              :                                      GO TO 30
*         :... RIGHT BRACKET    ]
*         :    END OF FUNCTION ARGUMENTS
*         :         :                                     290   IF (AND(TEST,113) .NE. 0) CALL FMLERR($990,ITC,1,1)
*         :         :                                           BRT = BRT-1
*         :         :                                           IF (BRT .LT. 0) CALL FMLERR($990,ITC,1,4)
*         :         :                                           TEST=4
*         :         :                                           CALL STKOUT($990,17)
*         :         :                                           IF (ITYP(NP) .NE. 16) CALL FMLERR($990,ITC,1,1)
*         :         :                                           I=D(NP)
*         :         :                                           J=NP
*         :         :                                           NP=LAST(NP)
*         :         :                                           CALL RMOVF1(J)
*         :         :                                           J=ITYP(NP)
*         :         :                                           IF (J .LT. 21) CALL FMLERR($990,ITC,1,4)
*         :         :                                           IF (J .EQ. 24) ITYP(NP)=I+24
*         :         :                                           IF ((J .EQ. 24) .AND. ((I+24) .GT. 31))
*         :         :                                          1    CALL FMLERR($990,D(NP),1,3)
*         :         :                                           IF (J .LT. 24) H2(NP)=I
*         :         :                                           CALL STKOUT($990,21)
*         :         :                                           GO TO 30
*         :... RIGHT PARENTHESIS   )
*         :         :                                     300   IF (AND(TEST,113) .NE. 0) CALL FMLERR($990,ITC,1,1)
*         :    CHECK IF THIS IS AN END OF SUBSCRIPT
*         :    OR THE END OF A SUBEXPRESSION
*         :    :                                                PAR = PAR-1
*         :    :                                                IF (PAR .LT. 0) CALL FMLERR($990,ITC,1,4)
*         :    :                                                CALL STKOUT($990,18)
*         :    :                                                IF (ITYP(NP) .NE. 17) CALL FMLERR($990,ITC,1,4)
*         :    :                                                IF (D(NP) .NE. 0) GO TO 310
*         :    :... END OF SUBEXPRESSION,
*         :    :    REMOVE MATCHING '('
*         :    :         :                                      I=NP
*         :    :         :                                      NP=LAST(NP)
*         :    :         :                                      CALL RMOVF1(I)
*         :    :         :                                      TEST=4
*         :    :         :                                      GO TO 30
*         :    :... END OF A SUBSCRIPT LIST
*         :         CHECK AND PACK SUBSCRIPTS
*         :              :                                310   TEST=2
*         :              :                                      N2=D(NP)
*         :              :                                      IF (N2 .GT. 4) CALL FMLERR($990,N2,0,5)
*         :              :                                      N3=0
*         :              :                                      SBC=SBC-1
*         :              :                                      IF (SBC .EQ. 0) SB= .FALSE.
*         :              :                                      DO 320 KK=N2-1,0,-1
*         :              :                                      K=NEXT(NP)
*         :              :                                      IF (ITYP(K) .NE. 0) CALL FMLERR($990,D(K),2,13)
*         :              :                                      IF (D(K) .LT. 0 .OR. D(K) .GT. 511) CALL FMLERR($990,D(K),0,15)
*         :              :                                      FLD(9*KK,9,N3) = D(K)
*         :              :                                320   CALL RMOVF1(K)
*         :         CHECK IF THIS IS THE END OF
*         :         TRANSLATION (ISW=2)
*         :         :                                           IF ((ISW .EQ. 2) .AND. (.NOT. SB)) GO TO 330
*         :         :... NO, GO BACK TO VARIABLE PART
*         :         :    TO GET THE VALUE OF THE
*         :         :    SUBSCRIPTED VARIABLE
*         :         :         :                                 J=NEXT(NP)
*         :         :         :                                 N1=D(J)
*         :         :         :                                 K=NP
*         :         :         :                            NP=LAST(NP)
*         :         :         :                            CALL RMOVFN(K,J)
*         :         :         :                            GO TO 140
*         :         :... YES, RETURN FOR ISW=2
*         :                   :                       330  CALL IFREEQ(NPO)
*         :                   :                            RETURN
*         :... MINUS   -
*         :    SET 'NEG= AND LINK IN +
*         :         :                                 340  NEG= .TRUE.
*         :... PLUS    +
*         :    IS IT UNARY OR BINARY
*         :    :                                      350  IF (AND(TEST,94) .NE. 0) GO TO 360
*         :    :... UNARY PLUS OR MINUS
*         :    :         :                                 CALL STKOUT($990,18)
*         :    :         :                                 TEST=16
*         :    :         :                                 GO TO 30
*         :    :... BINARY + -
*         :              :                            360  J=18
*         :    COMMON PART FOR BINARY OPERATORS
*         :         :                                 370  I=2
*         :         :                                 380  IF (AND(TEST,113) .NE. 0) CALL FMLERR($990,ITC,1,1)
*         :         :                                      TEST=16
*         :         :                                      CALL STKOUT($990,J)
*         :         :                                      NP=ILINK1(NP,J,I)
*         :         :                                      GO TO 30
*         :... MULTIPLICATION   *
*         :         :                                 390  J=19
*         :    GO TO BINARY OPERATOR
*         :         :                                      GO TO 370
*         :... EXPONENTIAL   **
*         :         :                                 400  J=20
*         :    GO TO BINARY OPERATOR
*         :         :                                      GO TO 370
*         :... DIVISION   /
*         :         :                                 410  I=-2
*         :         :                                      J=19
*         :    GO TO BINARY OPERATOR
*         :    SECOND ENTRY
*         :         :                                      GO TO 380
*         :... LEFT PARENTHESIS   (
*         :         :                                 420  IF (AND(TEST,78) .NE. 0) CALL FMLERR($990,ITC,1,1)
*         :         :                                      PAR = PAR+1
*         :         :                                      TEST=32
*         :         :                                      INEG = 3
*         :    GO TO CHECK FOR -1 FACTOR
*         :         :                                      GO TO 500
*         :         :                                 425  NP=ILINK1(NP,17,0)
*         :         :                                      GO TO 30
*         :... COMMA
*         :         :                                 430  IF (AND(TEST,113) .NE. 0) CALL FMLERR($990,ITC,1,1)
*         :         :                                      TEST=32
*         :         :                                      CALL STKOUT($990,18)
*         :         :                                      D(NP)=D(NP)+1
*         :         :                                      GO TO 30
*         :... EQUAL SIGN   =
*         :         :                                 440  IF (AND(TEST,117) .NE. 0) CALL FMLERR($990,ITC,1,1)
*         :         :                                      TEST=1
*         :         :
*         :         :
*         :         :
*         :    GET AND CHECK VARIABLE FOR ASSIGN
*         :    STATEMENT SAVE INFO. IN NN1,NN2,NN3
*         :         :
*         :         :
*         :         :
*         :         :
*         :         :
*         :         :
*         :         :
*         :         :
*         :         :
*         :         :
*         :         :
*         :         :
*         :         :
*         :         :
*         :    GO BACK TO TRANSLATE EXPRESSION
*         :         :
*         :...  SEMICOLON   ;
*         :...  APOSTROPHE   '
*              GO TO END OF TRANSLATION
*                   :
END OF LOOP

END OF TRANSLATION
        :
        :
        :
        :
     RETURN FOR 'ISW' = 1 AND 3
        :
        :
     TRANSFER ASSIGNED VARIABLE INFORMATION
        :
        :
        :
     RETURN FOR 'ISW' =0
        :
CHECK IF -1 FACTOR MUST BE INSERTED
     INDICATED BY 'NEG'
     :... NO, GO TO RETURN TO CALLING PLACE
     :         :
     :... YES, LINK IT IN
               :
               :
               :
          RETURN TO CALLING PLACE
               :
ERROR RETURN
     :
     :
```

```
      IF (EQL .OR. SB) CALL FMLERR($990,ITC,1,1)
      EQL= .TRUE.
      CALL STKOUT($990,18)


      IF ((ITYP(NP) .NE. 16) .OR. (D(NP) .NE. 1) .OR. (LAST(NP) .NE.  ))
     1    CALL FMLERR($990,ITC,1,1)
      KK=NEXT(NP)
      IF (ISW .EQ. 1) GO TO 450
      IF ((KK .EQ. 0) .OR. (ITYP(KK) .LT. 6) .OR. (ITYP(KK) .GT. 7))
     1    CALL FMLERR($990,ITC,1,1)
      NN1=D(KK)
      NN2=0
      IF (ITYP(KK) .EQ. 6) GO TO 450
      NN3=NEXT(KK)
      NN2=ITYP(NN3)-7
      NN3=D(NN3)
450   CALL IFREEO(KK)
      NEXT(NP)=0

      GO TO 30



460   IF (AND(TEST,113) .NE. 0) CALL FMLERR($990,ITC,1,1)



      IF (PAR .NE.0 .OR. BRT .NE.0) CALL FMLERR($990,'()  []',1,4)
      CALL STKOUT ($990, 18)
      IF ((ITYP(NP) .NE. 16) .OR. (LAST(NP) .NE. 0))
     1         CALL FMLERR($990,ITC,1,1)

      IF (ISW .NE. 0) RETURN
      IF (.NOT. EQL) CALL FMLERR($990,ITC,1,1)

      N1=NN1
      N2=NN2
      N3=NN3

      RETURN



500   IF (.NOT. NEG) GO TO 510

      NP = ILINK1(NP,19,2)
      CALL ILINK1(NP,0,-1)
      NEG = .FALSE.

510   GO TO (50,285,425), INEG

990   CALL IFREEO(NPO)
      RETURN 1

      END
```

APPENDIX B—PRINTED SUBPROGRAM: EXAMPLE 2

```
******************************************************
             TITLE
MAIN PROGRAM FOR INTERACTIVE FORMAL SYSTEM
                                                                PARAMETER IDIM = 10
                                                                DIMENSION IN(14), INN(14), ITAB(IDIM)
                                                                EQUIVALENCE (IN(2), INN(1))
                                                                DATA INN(14) / ';' /
                                                                DATA ITAB /'READ  PRINT DUMP  ERASE OPTIONCOMMEN'
                                                               + ,'ROLOUTNCOUNTSAVE  RESET '/
******************************************************
             SEQUENCE CHART
INITIALIZE BY CALLING FMLOPT
     :                                                    99    CALL FMLOPT ('INT;',0)
LOOP TO GET NEXT INPUT LINE
*    READ LINE
*        :                                                110   READ 100, END=200, IN
*        :                                                100   FORMAT (13A6,A2)
*    IF IT STARTS WITH 'C ' (COMMENT), GO TO GET NEXT
*    LINE
*        :                                                      IF (FLD(0,12,IN(1)) .EQ. 1005K) GO TO 110
*    IF IT STARTS WITH 'P ' (PRINT), GO TO 'P ' ENTRY
*        :                                                      IF (FLD(0,12,IN(1)) .EQ. 2505K) GO TO 22
*        :                                                      J = 0
*    LOOP TO GET STATEMENT TYPE IN J
*    *                                                          DO 111 I = 1,IDIM
*    *                                                    111   IF (IN(1) .EQ. ITAB(I)) J = I
*    END OF LOOP
*    J=0, IT IS AN ASSIGN STATEMENT
*        :                                                      IF (J) ,60,
*        :                                                      GO TO (121, 121, 121, 120, 120, 110, 120, 121, 120, 120), J
*    REPRINT ERASE,OPTION,ROLOUT,SAVE AND RESET
*    STATEMENTS
*        :                                                120   CONTINUE
*        :                                                      PRINT 101, IN
*        :                                                101   FORMAT (XA6,':',13A6)
*    BRANCH BY TYPE
*    :                                                    121   GO TO (1, 2, 3, 4, 5, 110, 7, 8, 9, 10), J
*    :... READ STATEMENT
*    :         :                                          1     CALL FMLIO1 (INN,0)
*    :         :                                                GO TO 110
*    :... PRINT STATEMENT
*    :         :                                          2     CALL FMLIO2 (INN,0)
*    :         :                                                GO TO 110
*    :    'P ' = PRINT
*    :         :                                          22    FLD(0,6,IN(1)) = 0505K
*    :         :                                                CALL FMLIO2 (IN, 0)
*    :         :                                                GO TO 110
*    :... DUMP STATEMENT
*    :         :                                          3     CALL ONDMP
*    :         :                                                K = 'P'
*    :         :                                                IF (INN(1) .NE. ' ') K = 0
*    :         :                                                CALL DUMP(K)
*    :         :                                                CALL OFFDMP
*    :         :                                                GO TO 110
*    :... ERASE STATEMENT
*    :         :                                          4     CALL FMLERS (INN,0)
*    :         :
*    :... OPTION STATEMENT
*    :         :
*    :         :
*    :... ROLOUT STATEMENT
*    :         :
*    :         :
*    :... NCOUNT STATEMENT
*    :         :
*    :         :
*    :... SAVE STATEMENT
*    :         :
*    :         :
*    :... RESET STATEMENT
*              :
*              :
*    ASSIGN STATEMENT
*         :
*         :
*         :
*         :
END OF FILE READ - STOP
*    :
*******************************************************
```

```
      GO TO 110

5     CALL FMLOPT (INN, 0)
      GO TO 110

7     CALL FMLOUT (INN,0)
      GO TO 110

8     CALL COUNT
      GO TO 110

9     CALL FMLSAV (INN)
      GO TO 110

10    CALL FMLRES (INN)
      GO TO 99

60    PRINT 102, IN
102   FORMAT (X14A6)
      CALL FMLASG (IN,0)
      GO TO 110

200   STOP

      END
```

APPENDIX C—PRINTED SUBPROGRAM: EXAMPLE 3

```
•••••••••••••••••••••••••••••••••••••••••••••••••           CMMN PROC
             TITLE
COMMON DATA STRUCTURE FOR FORMAL SYSTEM
•••••••••••••••••••••••••••••••••••••••••••••••••
             DATA STURCRUE
ARRANGED IN 3 LABELED COMMONS
USED AS PROCEDURE, INCLUDED IN OTHER SUBPROGRAMS
                                                                 IMPLICIT INTEGER(A-Z)
1. LINKED STORAGE AREA                                           PARAMETER ERROR = ERRERR
•    THE CORRESPONDING C(I)-D(I) WORDS ARE ALWAYS
•    USED IN PAIRS FOR STORING AN ITEM.
•    THE DIMENSION OF C-D, CDIM, MAY BE CHANGED
•    DURING INSTALLATION.
•    FIELDS IN THE C-D WORDS DEPEND ON THE USAGE,
•    THEY ARE DEFINED BY PROCEDURE 'PWORD', GENERALLY,
•    THE LAST 15 BITS IN C IS USED FOR LINKAGE OF
•    LINEAR ARRAYS.
•
•                                                                PARAMETER CDIM = 2048
•                                                                COMMON /FMLCM2/ C(CDIM)
•                                                                COMMON /FMLCM3/ D(CDIM)
2. COMMON BLOCK FOR INDIVIDUAL POINTERS AND SWITCHES
•                                                                COMMON /FMLCM1/
•... FREE (AVAILABLE) STORAGE IN C-D
•         C(NXNXNX) = FIRST
•         C(ILILIL) = LAST LOCATION
•         THE LINEAR ARRAY IS LINKED IN THE
•         LAST 15 BITS OF THE C-WORDS.
•             !                                                 +    NXNXNX, ILILIL
•... SYMBOL TABLE WITH TREE STRUCTURE IN 4 LEVELS
•         STORED IN C-D AREA. FIELDS IN THE C-WORD:
•             ITYPB - LAST - NEXT
•         NS = FIRST ENTRY IN C(NS)-D(NS)
•         :                                                     +    NS,
•         NSB = SUBROUTINE LEVEL POINTER
•         !    SUBPROGRAMS ARE IN ALPHABETIC ORDER
•         !                                                     +    NSB,
•         !... ITYPB(NSB) = 0
•         !... D(NSB) = ALPHANUMERIC NAME OF THE
•         !    SUBROGRAM
•         !... NSY = LAST(NSB) POINTER TO SYMBOL ENTRY
•         :    !    SYMBOLS ARE IN ALPHABETIC ORDER
•         :    :    NSY1 = POINTER TO PRECEEDING SYMBOL
•         !    :    ENTRY
•         :    :                                                +    NSY,NSY1,
•         !    !... D(NSY) = ALPHANUMERIC NAME OF THE
•         !    :    SYMBOL
•         !    !... ITYPB(NSY)= TYPE OF SYMBOL,
•         :    !    !    SEE TABLE I.
•         !    !    !... >31, INDIRECT REFERENCE
•         !    :    !    LAST(NSY) POINTS TO AN OTHER
•         !    :    !    SYMBOL
•         !    :    !
•         !    !    !... SECOND AND THIRD BIT = 11
•         !    !    !    SUBSCRIPTED VARIABLE
```

```
*         :    :    :    NSU = LAST(NSY), POINTER TO
*         :    :    :    :    SUBSCRIPT ENTRY
*         :    :    :    :    SUBSCRIPTS ARE ORDERED BY
*         :    :    :    :    NUMBER OF SUBSCRIPTS, AND
*         :    :    :    :    BY ACTUAL SUBSCRIPTS, NSU1
*         :    :    :    :    = POINTER TO PRECEEDING
*         :    :    :    :    SUBSCRIPT ENTRY
*         :    :    :    :                                     +    NSU,NSU1,
*         :    :    :    :... D(NSU) = NUMERIC SUBSCRIPT,
*         :    :    :    :    SEE TABLE II,
*         :    :    :    :... FIRST 3 BITS OF C(NSU)=
*         :    :    :    :    :... =110, UNASSIGNED,
*         :    :    :    :    :    INDIRECTLY REFERENCED
*         :    :    :    :    :         LAST(NSU) POINTS
*         :    :    :    :    :         TO OTHER SYMBOL
*         :    :    :    :    :         ENTRY
*         :    :    :    :    :... =111, LAST SUBSCRIPT
*         :    :    :    :    :    ENTRY,
*         :    :    :    :    :         LAST(NSU) POINTS
*         :    :    :    :    :         BACK TO ITS
*         :    :    :    :    :         SYMBOL = NSY
*         :    :    :    :    :... =010, NORMAL ENTRY
*         :    :    :    :    :         LAST(NSU) POINTS
*         :    :    :    :    :         TO EXPRESSION
*         :    :    :    :    :         VALUE = EPTR
*         :    :    :    :... NEXT(NSU) = FORWARD LINK TO
*         :    :    :         NEXT SUBSCRIPT,ZERO FOR
*         :    :    :         THE LAST ONE
*         :    :    :... OTHERS, LAST(NSY) = EPTR
*         :    :         EPTR = EXPRESSION POINTER
*         :    :         :                                     +    EPTR,
*         :    :         :... ITYPB(EPTR) = 16
*         :    :         :... LAST(EPTR) = 0
*         :    :         :... H2(EPTR) = NUMBER OF
*         :    :         :    EXPRESSIONS (FOR LISTS)
*         :    :         :... H1(EPTR) =
*         :    :         :    :... =0, EXPRESSION IS IN
*         :    :         :    :    CORE
*         :    :         :    :... NOT ZERO, EXPRESSION
*         :    :         :              IS ON DRUM
*         :    :         :              INDEX = H1(EPTR)
*         :    :         :... NEXT(EPTR) FORWARD LINK
*         :    :              TO THE LINEARLY STORED
*         :    :              EXPRESSION WHEN IT IS
*         :    :              ON CORE,THE CONSECUTIVE
*         :    :              ENTRIES ARE ACCORDING
*         :    :              TO TABLE III,
*         :    :... NEXT(NSY) = FORWARD LINK TO NEXT
*         :         SYMBOL, ZERO FOR LAST ONE
*         :... NEXT(NSB) = FORWARD LINK TO NEXT
*              SUBPROGRAM, ZERO FOR LAST ONE
*
*... TEMPORARY VARIABLES FOR
*         N1 = NAME OF A VARIABLE
*         N2 = NUMBER OF SUBSCRIPTS
*         N3 = SUBSCRIPT WORD
*         IY = 3 OR 1 FOR SUBSCR. OR NOT
*                                                                      +     N1,N2,N3,IY,
*...* OPTION SWITCHES
*           XQTOPT = OPTION WORD FROM WAIT STATEMENT
*           PRODEX = EXPAND POWERS OVER PRODUCT
*           INTGSW = EVALUATE INTEGER VALUED FUNCTIONS
*           MATHSW = EVALUATE MATHEMATICAL FUNCTIONS
*           EXPDSW = USE DISTRIBUTIVE LAW
*           POWER  = EXPAND SUMS RAISED TO POS. INTEGERS
*           BASE   = 0,1,2,3 FOR BASE(0),(2),(10),(E)
*                                                                      +     XQTOPT,PRODEX,INTGSW,MATHSW,EXPDSW,POWER,BASE,
*...* MISCELLANEOUS
            SIMPSW = USED BY STOUT ROUTINE FOR RECURSIVE
                     SIMPLIFICATION
            BITSW  = USED BY STOUT ROUTINES
            IOUNIT = I/O UNIT NUMBER IF I/O STATEMENTS
            FTRARG = NUMBER OF FORTRAN TYPE ARGUMENTS
            DEFARG = NUMBER OF ARGUMENTS IN A DEFINED
                     FUNCTION
            DEFFUN = 1 IF DEFINED FUNCTION, 0 FOR VARIABLE
            NK = START OF ARGUMENT CHAIN IN C-D FOR LIST
                 OF VARIABLES
            CBUF = I/O BUFFER
            NP = PUSH-DOWN STACK POINTER IN C-D AREA
                                                                       +     SIMPSW,BITSW,IOUNIT,FTRARG,DEFARG,DEFFUN,NK,CBUF,NP
******************************************************                     LOGICAL INTGSW,MATHSW,POWER,SIMPSW,BITSW,PRODEX
                                                                           REFERENCES ON
                                                                     END
```

APPENDIX D—DEFINITION OF INITIAL SEQUENCE CHART

Coding Form

The coding form is divided into three fields: Field 1 consists of one character, the general directive for input; field 2 contains special directives for flowchart elements and a label for program statements; field 3 contains the text.

An initial program is illustrated below:

```
T     EXPRESSION TRANSLATION
S     INITIALIZE
0D    LOOP TO PROCESS CONSECUTIVE SYMBOLS
1D    |     BRANCH BY TYPE OF SYMBOL
2B    |     ├INTEGER
2B    |     ├REAL
2B    |     ├IDENTIFIER
2BE   |     └SPECIAL CHARACTER
0     END OF LOOP
0     END OF TRANSLATION
            END
```

Input Form

The actual input does not contain the lines; the text is left adjusted in field 3:

```
T     EXPRESSION TRANSLATION
S     INITIALIZE
0D    LOOP TO PROCESS CONSECUTIVE SYMBOLS
1D    BRANCH BY TYPE OF SYMBOL
2B    INTEGER
2B    REAL
2B    IDENTIFIER
2BE   SPECIAL CHARACTERS
0     END OF LOOP
0     END OF TRANSLATION
      END
```

Output Form

The initial program can be listed with line numbers as follows:

```
     *******************************************
1 =           EXPRESSION TRANSLATION
     *******************************************
              SEQUENCE CHART
2 = INITIALIZE
```

```
  3 = LOOP TO PROCESS CONSECUTIVE SYMBOLS
  4 = :    BRANCH BY TYPE OF SYMBOL
  5 = :    : . . . INTEGER
  6 = :    : . . . REAL
  7 = :    : . . . IDENTIFIER
  8 = :    : . . . SPECIAL CHARACTERS
  9 = END OF LOOP
 10 = END OF TRANSLATION
 11 =                                              END
```

APPENDIX E—EXAMPLE OF AN UPDATING PROCEDURE

```
+1
            SUBROUTINE EXPRES (*, ISW)
+R7D        |
1B          |-IDENTIFIER NOT TERMINATED BY ( OR [
1B          |-IDENTIFIER TERMINATED BY (
1BE         L-IDENTIFIER TERMINATED BY [

+R8D        |
1B          |- -
            |  NEG = .TRUE.
1B          |- +
            |  J = 18
1B          |- *
            |  J = 19
1B          |- **
            |  J = 20
1BE         L- /
               J = 19
               I = -2
```

Note that the '+' is an insertion directive. The number following + indicates the line where the insertion is to be done. 'R' indicates that the levels of lines following to be inserted are defined relative to the line where the insertion occurs.

DISCUSSION

MEMBER OF THE AUDIENCE: I notice that you have many comments noted through there. It seems to be about a two-to-one comment per statement. Is that about correct?

MESZTENYI: It depends on the program. It depends on the language, too. The comments should be semantic, not repeated as an equation.

MEMBER OF THE AUDIENCE: Do you think that some of the discussions about what we can get out of the compiler would fall into this?

MESZTENYI: I would like to have the compiler in the subroutine. I would like to do that, but I would start here from the development point first, because this is where one defines the program first.

MEMBER OF THE AUDIENCE: It seems that the compiler could give you certain information, and you could add some personal comments and have better descriptive material. Is that true?

MESZTENYI: It depends on what standpoint you look at. As I look at it, I want an overall view from the beginning. Before I finish the program, I might want to give the specification a bigger flowchart type of definition that could be used right away.

MEMBER OF THE AUDIENCE: You are trying to get the flavor of the program that you are working on for a certain purpose. The compiler will only come out with standard words for any program. The compiler does not know what your program is, but you do. With personal comments added to the program, what you have would provide additional information.

MESZTENYI: I find it is hard for programmers to add something after they have written the program. When they write, they do not mind writing down their comments.

MEMBER OF THE AUDIENCE: I am working from the viewpoint that we now have difficulty at times getting any comments in, and if we provided a lead into the comments and they went down the list and it did not make too much sense to them from a general viewpoint, that they could add these rather well.

MESZTENYI: I agree that they could, and this is actually what is now done. I added this myself.

The other part I would like to focus on a little bit is the programming part. If you start from the sketch with those lines coming down and write, you make the programmer apply a little discipline to the subject of program placement. For example, I try to avoid any GO TO unless it is some kind of loop structure. I try to avoid going back. I find a loop for each logic curve that I process, but it is not a DO statement, and I jump directly back to the beginning. It probably would have been much nicer documenting it to go to the end of this loop and comment it, which goes back and gets the next one. In this way it forces the programmer to do a documented description because it is very hard to document a graph that points out the actual information. The text or the description of the program is sequential, but semantically it is a graph. A tree, which is sort of in-between, is much easier to represent. You have cross-references, but the form is still a tree, and this is what I tried to simulate.

MEMBER OF THE AUDIENCE: I think the speaker is trying to get the programmer to write down what is being accomplished and when. Once in the right-hand side, the language does not really matter. He is trying to read narrative text so that you get some concept of when things happen and what really is happening because the specification of the problem is written in a narrative form. He does that rather than deduce what was done from how something is being done. I do not think a programmer is going to do that very well because he is so involved in the mechanics that he cannot get out of them.

MEMBER OF THE AUDIENCE: It seems to me that here is a case where we can go from the rationale of a subroutine and in an automated way feed in the programming language statements. Is this what you had in mind? I could see how you actually tried to develop your subroutine. I can see how you can start with the rationale of the subroutine first and then by using the type of coding that you did, you could automatically call for the appropriate programming language statements.

MESZTENYI: Not automatically. I certainly think of more than just the semantic type of description that I want to accomplish. What I want to accomplish eventually is the statements.

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.

PRECEDING PAGE BLANK NOT FILMED

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.

PANEL DISCUSSION

MEMBER OF THE AUDIENCE: What recommendations or conclusions can we come to?

PANEL MEMBER: I would like to put in another pitch for a little more standardization. It seems to me that we will never be able to automate unless we standardize a few things. I think the FORTRAN standards have helped immensely, and I would like to see standards developed in other areas, including documentation.

PANEL MEMBER: I plan to submit formally my proposal for work adding to the FORTRAN compiler. I notice several people have made comments from the audience about changing the compiler and making it more efficient. I think this would be a good opportunity to perhaps form a committee for making recommendations for the next compiler. I think that now that we are all enthusiastic, we should try to do something effective.

PANEL MEMBER: Let me make a pitch for not having standards, especially in documentation. I do not think the problem is one of standards in the sense that it seems to be used here. There are already flowcharting standards and anyone can put together standards for decision tables and all the other tools that we have talked about.

The standards we should talk about, it seems to me, are professional standards. A mathematician writing a paper in a journal follows very rigid standards. There are no questions about documentation. If the man cites a theorem, he has to give a reference or prove it. This is not because there is a written standard that says you have to prove all theorems. It is part of his professional standards. If programmers are going to call themselves professionals, they must accept certain professional standards. Standards for language are fine, but standards for documentation should simply be accepted as part of a man's work. The fact that we are talking about documentation separately shows that programmers are not yet completely professional. I also do not understand why there should be such a problem about motivation. One does not talk about the motivation of a mathematician or physician to do a good job. This too should be part of the professional standard of a programmer.

One of the things I hope we will emphasize is the place of management in all this; the other thing is working at different approaches. We should look at programming languages and see how to make them more self-documenting. I think we should also look at the way we organize programming teams. Dr. Mills has suggested organizing programming teams along surgical team lines. One or two programmers do the programming, one man does the documentation, and one does all the job control. There may be human problems involved in this, but this solution should be studied.

MEMBER OF THE AUDIENCE: Is it technically possible to develop a system to completely automate documentation?

MEMBER OF THE AUDIENCE: I would like to add to that question: Is it possible to develop standards for documenting and cover all special programs, or is more than one set of standards needed, each set satisfying a specific class of programs and a specific class of applications? Maybe scientific applications of programming need a different set of standards than the data processing applications do? Would one set of standards satisfy all the different applications that we have today?

MEMBER OF THE AUDIENCE: I would like to object to the idea of standards in documentation. I feel that it is another indication that programming is not yet professionalized. It is the responsibility of the people who are buying programs to realize that documentation ought to be part of the job and not an extra cost option.

MEMBER OF THE AUDIENCE: Could I get an answer to my question? Is it technically possible to completely automate program documentation?

PANEL MEMBER: I would vote no.

MEMBER OF THE AUDIENCE: May I modify that question? To what degree can it be automated?

PANEL MEMBER: The question really has no answer because you have not told me what kind of documentation you want.

MEMBER OF THE AUDIENCE: Adequate documentation.

PANEL MEMBER: But to what, or related to what, for whom? A lot of documentation can be automated, but not all of it. As I said, it cannot be automated unless you specify the kind of documentation you need.

MEMBER OF THE AUDIENCE: Adequate documentation, so somebody else can use the program.

PANEL MEMBER: I think you could produce documentation in an automated fashion right now that would let someone do most of the things that should be done with a program. It would take some work, but it could be done.

PANEL MEMBER: In my opinion, I think documentation is indicative of what has happened with the computer. It has gone from completely automatic programming to a programming that is an interaction between man and machine. As I see it, documentation is similar in that there are certain guidelines, contents, or content descriptions of documents that you want, and there is a lot of automatic help to do it, whenever the proper input and the proper output are defined. But even that might leave something out. You do not want to standardize to the point that human decisions are left out. So documentation would be automated only to the extent that it would be an aid in a human process.

MEMBER OF THE AUDIENCE: Do you think that the cost of documentation can be cut by automation? What help can we give the programmer in this area?

PANEL MEMBER: I think that it is definitely possible to do something. I think that we have seen examples of what has been done. A most interesting point is that some of the work on this problem has been around for a while. Some of the loader programs in the 7094 had abilities that are not in third-generation software. The insert file facility, for example, is quite a powerful tool, but is not considered a standard part of a language compiler.

MEMBER OF THE AUDIENCE: I think that until the kind of information needed to support a certain kind of activity is identified, there can be no real talk about cutting the costs of documentation.

We ought to have an idea about the kind of information that the man at the console needs. We ought to talk to him and find out exactly what kind of information he needs and thinks he needs.

MEMBER OF THE AUDIENCE: It is true that this has not been done for the industry as a whole, but there are certain companies that have done this for themselves and do have well-documented programs. The parts that can be automatically documented, are, and everybody is satisfied.

I think the problem is not that we do not have the tools, but that we do not know what to do with the ones we have now. I also believe that what is needed are professional standards that tell us what constitutes a good program and what information should be available to somebody using that program. However, no one seems to want to work on that, so there seems to be little hope that we will solve this problem.

* U. S. GOVERNMENT PRINTING OFFICE: 1972–735-966/535