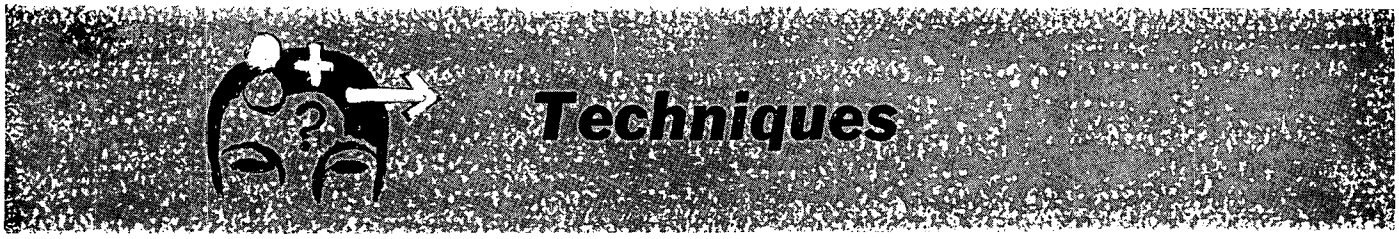

C. L. McCARTY, JR., Editor

Design and Implementation of a General-Purpose Input Routine

A. HASSITT
University of California, San Diego, La Jolla, California

A general-purpose input routine is discussed and advocated for FORTRAN. The philosophy of such programs is examined and exemplified.

1. Introduction

There are many ways of writing a program to do a particular job. When the job is not particular but has some general purpose such as "read data" or "translate algebraic formulas" or "read control word" or "read next input," then there are many ways of even starting to think about writing the program. In such a situation, the first approach is to give a more exact description of what is to be done. This paper is concerned with the problem of getting data and control information into computers. It is assumed that the datum is in manuscript form and was not a product of some previous computer operations. Currently available routines either make the data suffer or they make the user suffer. For example, the format-free read statement in the Oak Ridge ALGOL Compiler [3] restricts the data to numeric, whereas the formatted input in the same compiler [3] or in FORTRAN [8] places an unnecessary burden both on the originator of the data and on the programmer.

Section 2 describes the design criteria of an input routine that is easy to use, that reads all types of data and control information, and that reads correctly anything that can be read by the human eye. Section 3 gives a user's description of a FORTRAN implementation of such a routine. Following sections describe some uses of the routine and finally, there is a description of the routine itself. The input routine is written in the CDC FORTRAN-63 language [4], but since the syntax is expressed in the form of a decision matrix, the routine is easy to understand, and could be copied into any other compiler language. The only difficulty of copying the routine into ALGOL [3] or IBM FORTRAN [8] is that these languages lack some of the features of FORTRAN-63.

2. Design Criteria

A general-purpose input routine should satisfy the following criteria.

(a) It should read words and symbols in addition to its prime purpose of reading numbers.

(b) It should consider the printed appearance of the input and should attach the normal meaning which this appearance implies. For example,

$$\text{ALPHA}=61.2\text{Y}=56\quad 99$$

can be read by eye as word ALPHA, symbol =, number 61.2, word Y, symbol =, number 56, number 99. The input routine should not insist on a space between the A and the = or the = and the 6 or the 2 and the Y, but if these spaces exist, then they should have no effect. For example,

$$\text{ALPHA} = 61.2 \quad \text{Y} = 56 \quad 99$$

has the same meaning as the previous example.

(c) The routine should be basic. For example, the routine LIST (see Sec. 4 and [1]) has many uses, but if a good basic routine is available then the average programmer can easily write his own LIST routine.

(d) The routine should be easy to use by the inexpert programmer and powerful enough for most of the needs of the expert.

The routine which is described later has the following incidental virtues.

(e) It can be used on character-by-character (paper tape) or unit-record (card, magnetic tape) input devices.

(f) It is easy to modify. For example, one could easily change from the FORTRAN notation 12.3E4 to a notation like 12.3,4 or even 12.3+4.

The importance of rule (b) cannot be overemphasized. Sight verification of data is used to check not only keypunching [7] but also the data. If the data input to a program is well designed then a number of sources of error are eliminated. It is a curious fact that the FORTRAN compiler ignores blanks in statements, so that

$$\text{DO 100 I = 1,6}$$

can be punched as

$$\text{DO100I=1,6}$$

without causing any trouble, but the same sort of reasonable change to FORTRAN data would usually cause chaos. Input routines for paper tape input [2, 10] usually follow rule (b) but ignore the problem of nonnumeric data or control information. Purely numeric datum is quite adequate for small programs, but in programs with complex

data requirements, the data presentation must be well designed and this requires the use of mnemonic words.

3. FORTRAN Routine

The following routine is the FORTRAN-63 language on the CDC 1604 computer. A similar routine could be used in any FORTRAN or ALGOL system. In the description, a reference is made to *card input.* The routine normally operates on *card images* from a magnetic tape; as will be seen, the routine is virtually independent of the input medium and indeed it could operate on a string of characters held in core store.

The routine has two main entry points. Assume for the moment that the cards contain only numeric data. CALL RDNUM(X) reads the next number from the input stream, converts it to floating-point binary form and puts it in X. CALL RDINT(I) reads the next number, converts it to a binary integer and puts it in I. The numbers may be punched anywhere on the card. A number is of the form

bb···b±dd···d.dd···dEbb···b±dd···dx

where

bb···b = none, one or more blanks
dd···d = none, one or more digits
± = + or − or may be omitted
. = decimal point or may be omitted.

The sequence dd···d.dd···d must contain at least one digit.

E denotes exponent with base 10. If E occurs it must follow the fractional part immediately with no intervening spaces. This rule is unfortunate but necessary as the whole exponent sequence Ebb···b±dd···d may be omitted.

x denotes the terminating character which is usually a blank but could be: (1) a letter, (2) a special character ($, =, etc.), (3) a second appearance of the character "." or character E, or (4) a + or − sign following a digit. If a number has been started then the end of a card acts as a terminating character. The end of a card is ignored (a new card is read in) if only blanks have appeared so far. A typical porton of a program might be

```
      CALL RDINT(I)
      DO 1 J = 1,I
1     CALL RDNUM(X(J))
```

which might operate on data such as

```
9    1.234   −0.16E 2   0.1   54   123456789123456789
0  −9        56E3       21.26
```

Notice that 0 or 0. or .0 or 0.0 (but not .) can be written and that 54 or 54. or 54.0 or .54E2 or .54E 2 (but not .54 E2) can be written.

This routine will also read words and special characters. A word is of the form

bb···b aa···ax

where

bb···b = none, one or more blanks
aa···a = one or more of the alphabetic characters A,B,C, ..., Z. If there are more than eight characters the ninth and subsequent characters are ignored.
x = any nonalphabetic characters.

A special character is none, one or more blank characters followed by $ or , or / or * or = or . or (or). Suppose the input card contains

ALPHA=61.2E3 C2 E5

then

```
      DO 1 J=1,7
1     CALL RDNUM (X(J))
```

would set X(1) = 5HALPHA, X(2) = 1H=, X(3) = 61.2E3, X(4) = 1HC, X(5) = 2.0, X(6) = 1HE and X(7) = 5.0.

In many contexts it is important to know whether the quantity just read is a number, a word or a special character. The routine has an alternative entry point.

CALL RDTYPE(K)

sets K = +1, 0, −1 depending on whether the last quantity read was a number, a special character or a word. Thus,

```
      DO 1 J=1,7
      CALL RDNUM (X(J))
1     CALL RDTYPE (L(J))
```

operating on the data given above would set X(1) = 5HALPHA, etc., and L(1) = −1, L(2) = 0, L(3) = +1 and so on. Notice that in the above example the space separating 2 and E is taken to mean that E is a word and not an exponent indicator.

4. Order Independent Input

Bailey, Barnett and Futerelle [1] have described some of the advantages of arranging the input data so that they are independent of the order on the cards. They give a routine called LIST which is useful for this purpose. There are a number of possible routines of this sort and [1] describes some of these possibilities. It would be impossible to provide a LIST routine to suit every possible use but we can show that it is a simple matter to write such routines, providing a good basic input routine is available. The following routine is similar to, but not identical with, the LIST routine by Bailey. We have had to avoid the difficulty (in a routine purely in FORTRAN) of having an unspecified number of arguments.

LIST can best be described by considering the following example:

```
DIMENSION L(4),X(4)
DATA (L=1HB,3HXYZ,1HP,5HALPHA)
EQUIVALENCE (X(1),B), (X(2),XYZ), (X(3),P),
  (X(4),ALPHA)
CALL LIST (L,X,4)
```

LIST will read cards that contain data such as

ALPHA=6.2 P=9 B=6 STOP

It will recognize words which were named in the L-vector and store the numbers which follow in the appropriate place in the X-vector. LIST will terminate when it encounters an unrecognized word

```
      SUBROUTINE LIST (L,X,N)
      DIMENSION L(100), X(100)
3     CALL RDNUM(I) $ DO 1 J=1,N
      IF (L(J)-I) 1,2
1     CONTINUE $ RETURN
2     CALL RDNUM(I) $ CALL RDNUM (X(J))
        GO TO 3 $ END
```

For this example, we have used that fact that CDC FORTRAN-63 allows several FORTRAN statements on one line providing they are separated by the $ symbol.

We could go on to refine this subroutine. For example, the equality sign can be made optional by replacing statement 2 by

```
2     CALL RDNUM(X(J)) $ CALL RDTYPE(I)
        IF (I) 2,2,3
```

or parenthetic comments can be allowed if such comments were surrounded by opening and closing brackets. To do this replace statement 3 by:

```
3     CALL RDNUM(I) $ IF(I-1H() 10,11
11    CALL RDNUM(I) $ IF (I-1H)) 11,10
10    DO 1 J=1,N
```

It is also possible to allow the use of brackets within brackets.

In reading a string of characters there is one case in which there are two possible interpretations. Consider the string ALPHA99 = 56.2 the first 6 characters can be read as: (1) word ALPHA, number 99, or (2) word ALPHA99. The routine RDNUM usually gives the first interpretation. It can be made to give the second interpretation by CALL RDMODE(2). This mode of interpretation stays set permanently but the mode of interpretation can be reset by CALL RDMODE(1).

5. Input of Simultaneous Equations

Suppose we have a routine which will solve a set of linear simultaneous equations; for example, we might wish to solve

$$\begin{aligned} X1 + 3.2X2 - X3 &= 4.5 \\ 19X1 + 21.2X3 &= 0 \\ 6X2 + 3X3 &= 7 \end{aligned}$$

It would be convenient and conducive to accuracy if the input could take precisely this form. Notice that (a) it should not be necessary to specify elements with zero coefficients; (b) X1, +X1 and −X1 should have their usual interpretation; it should not be necessary to write 1.0X1, +1.0X1, or −1.0X1; (c) it should not be necessary to specify the number of equations explicitly. The following routine will read such input, it will put the coefficients in array A and the right-hand sides in vector R.

```
      DIMENSION L(3), A(10,10), R(10)
      DATA (L= 1H+, 1H-, 1H=)
      DO 1 I = 1,10 $DO 1 J = 1,10
1     A(I,J)=0.0 $ J=1 $ NEQ=1
7     CALL RDNUM(W) $ CALL RDTYPE(I)
        IF(I) 2,3,4
4     CALL RDNUM(B) $ CALL RDTYPE(I) $ IF
        (I) 5,6,6
2     W=1.0
5     CALL RDINT(I)
8     A(J,I)=W $ NEQ=XMAXO(NEQ,I)
      GO TO 7
3     I=INDEX(W,L,3) $ GO TO (11,12,13,6), I
11    W=1.0 $ GO TO 4
12    W=-1.0 $ GO TO 4
13    CALL RDNUM(R(J)) $ IF(NEQ.GT.10) 21,22
22    J=J+1 $ IF(J.GT.NEQ) 23,7
```

The normal finishing point is 23. Finishing at statement 6 indicates a data error. Finishing at statement 21 indicates that an attempt is being made to read too many equations. The subroutine INDEX sets I = 1, 2, 3 or 4 depending on whether W = L(1) or L(2) or L(3) or none of them.

```
      FUNCTION INDEX(I,L,N)
      DIMENSION L(10) $ DO 1 J=1,N $ IF(I-L(J))
        1,2
1     CONTINUE $ J=N+1
2     INDEX=J $ END
```

This routine is similar to a sequence in the LIST routine.

It is obvious that writing this sort of sequence of FORTRAN statement takes longer than the simple

```
READ I, ((A(K,J),J=1,I),R(K),K=1,I)
```

but the gain in terms of accuracy and efficiency of data input more than compensate for any extra programming.

6. Further Considerations

Yarborough [13] has discussed two of the advantages of using alphabetic words with numeric data. First, the data can be split into convenient blocks, each block headed by its own code word; second, in a series of problems in which some blocks are fixed and some vary, the fixed-data are read in for one problem, then each subsequent problem is preceded by those parts of the data which have changed.

One of the most powerful uses of a flexible input routine is to make it control the operations of the main program. A routine similar to RDNUM was used in a large reactor code [5]. The first part of the data to the program specifies the size and composition of the reactor. Following this there is a series of control words which determine what the program will do with these data. Some of the control words and their meanings are:

FLUX	A estimate of the neutron flux is available from a previous problem; read this estimate.

DUMP	Save the results of this problem for later processing.
TAPE n	The next FLUX or DUMP word will use tape n.
UNLOAD n	Unload tape n.
PRINT n	Print results of type n.
ALTER n	Read a further series of items to alter certain parts of the data
ENTER	Do a multigroup diffusion calculation.
NEXT	Go on to do the next problem.

A typical control sequence would be:

```
TAPE 9 FLUX UNLOAD 9 ENTER
TAPE 10 DUMP ALTER.....ENTER
PRINT 2 UNLOAD 10 NEXT
```

Once having begun the process of dynamic control of the program, one might wish to take certain options depending on the course of the calculation. This was achieved in the reactor code by the word COMPILE, which was an instruction to the code to read in algebraic formulas. These formulas could process the results obtained so far and decide on the future course of action.

The routine RDNUM essentially reads one number. It is simple, but sometimes tedious, to build a nest of DO's around the RDNUM statement. There are various ways of alleviating this situation. Suppose we wish to do the following:

```
READ ((A(I,J), I=1,10), J=1,10), X,Y,Z
```

Then it can be done in various ways: (a) write the necessary coding using RDNUM; (b) read all the numbers into a vector array with a simple DO loop, then distribute the numbers by use of a DECODE statement; however, there are several practical difficulties in trying to do this; (c) the FORTRAN compiler has a mechanism for sorting out such read statements and it is possible to make use of this feature. It is a simple matter to alter the FORTRAN read statement so that READ FORMAT, LIST calls upon RDNUM for its numbers; we have done this for FORTRAN II [6]. The real solution to this problem is that the FORTRAN compilers should be changed so that the input-output list facility is available for general use. Such generated lists should also carry with them the mode of the argument.

7. Input Routine

The routine can be split into three stages: the scan, the syntax and the semantics. The scan is a character-by-character process. Let I be a counter whose value is initially set to 73. (It has this value before RDNUM is entered for the first time.) Then on entering the routine RDNUM, test I and if

I > 72	(a)	read a card image into array X, set I = 1, go to (c).
I = 72	(b)	set the current character equal to the blank character, set I = I + 1, go to (d).
I < 72	(c)	set the current character equal to the ith character from the array X, set I = I + 1, go to (d).
	(d)	the scan for one character is now complete.

In the first phase of the syntactical analysis we divide characters into one of the following eight categories:

1.	| 0 | 1 |....|9|	5.	| A| B| C|D| F| G|...| Z|
2.	| . |	6.	|∧|
3.	|+|−|	7.	| $ |=| (|) | , | * | / |
4.	| E|	8.	Illegal characters

where ∧ denotes the space symbol.

The second phase uses the decision-matrix method, which was suggested in this context by Vasilakos [12]. The decision matrix appears as Table 1.

The routine starts in stage 1; encountering a digit in stage 1 would set the stage counter to 5, a "." would set the stage to 3 and so on. A digit in stage 7 would set the next stage to 9. The letters A through G represent terminating stages.

A = string is illegal
B, C = string is a special character
D, E = string is a number
F, G = string is a word.

For example, ∧∧1E∧6∧ goes through stages 1, 1, 5, 7, 7, 9, E to indicate that a number has been read; 1EE goes through 5, 7, A and is illegal; ∧E6 goes through 1, 10, G to indicate a word (the word "E") has been read. With each exit, there are two possibilities: C, E and G indicate backspace one character (I = I − 1) prior to the next entry to RDNUM; B, D and F do not backspace. For example, if the string to be read is ∧E6.2∧ the routine comes out on a G-type exit and indicates that the word E has been read; the next time the routine is entered it will start scanning at the character 6 and go on to read the number 6.2.

The semantics are quite straightforward. For example, in stage 10 we find (see Figure 1):

```
      J = WORD.AND.7700 0000 0000 0000 B
      IF(J) 29,28
28    WORD = WORD*64
      WORD = WORD.OR. CHARAC
29    Go to read next character
```

TABLE 1. DECISION MATRIX FOR THE ROUTINE RDNUM

Type.... Stage	*Digit* 1	. 2	+− 3	*E* 4	*A···Z* 5	∧ 6	*Special* 7	*Illegal* 8
1	5	3	2	10	10	1	B	A
2	5	4	C	C	C	C	C	A
3	6	C	C	C	C	C	C	A
4	6	A	A	A	A	A	A	A
5	5	6	E	7	E	E	E	A
6	6	E	E	7	E	E	E	A
7	9	A	8	A	A	7	A	A
8	9	A	A	A	A	8	A	A
9	9	E	E	E	E	E	E	A
10	G	G	G	10	10	G	G	A

This is simply building up a word, allowing for the possibility that eight characters have already been read: initially WORD = 0, finally WORD = character string read, assuming that the string is a word. If a number is being read then the final result is built up in a manner which is standard in routines of this sort [11]. For example, if the input string is 12.345E6 then at the end of the scan there are three registers J, K, L with the values J = 12345, K = number of decimal places = 3, L = exponent = 6.

```
      SUBROUTINE RDNUM(ISIT)
      TYPE INTEGER STAGE,COLCOUNT,COLMAX,CHARAC,CHARTYPE
     X,OLDCHAR,RDBYTE      ,WORD,TESDC,EXPONENT,DIGITS,ESIGN,SIGN
      DIMENSION BUFFER(10),TYPETAB(4),DECISION(12),XPA(4),XPB(10),XPC(10
     1)
      DATA (COLCOUNT=74),(COLMAX=73),(PRINTING=1.0),(MASK=77B),
C     USES THE 1604 REPRESENTATION OF BCD CHARACTERS
     X (TYPETAB= 0111111111173000B,6755555555077000B,3555555555077000B,
     X3555545555027007B),(DECISION=8H532==1BA,8H54CCCCCA,8H6CCCCCCA,
     X8H6AAAAAAA,8H56E7EEEA,8H6EE7EEEA,8H9E8AA7AA,8H9EAAA8AA,8H9EEEEEEA,
     X8HGGG==GGA,8HGGG==GGA,8H=GG==GGA)
     X      ,(XPA=1.0,1.0E100,1.0E200,1.0E300),(XPB=1.0,1.0E10,1.0E
     X20,1.0E30,1.0E40,1.0E50,1.0E60,1.0E70,1.0E80,1.0E90),(XPC=
     X1.0,10.0,100.0,1000.0,10000.0,1.0E5,1.0E6,1.0E7,1.0E8,1.0E9)
      EQUIVALENCE (X,I)
      FLOATING=1.0 $ GO TO 1 $ ENTRY RDINT $ FLOATING=0.0
 1    STAGE=1 $ N=DIGITS=EXPONENT=WORD=TESDC=SIGN=ESIGN=0
 11   OLDCHAR=CHARAC    $ IF (COLCOUNT-COLMAX) 3,6,2
 6    CHARAC=1R       $ GO TO 7
 2    READ 500, (BUFFER(I),I=1,10)
 500  FORMAT(10A8)
      IF (PRINTING) 4,5
 4    PRINT 501,(BUFFER(I),I=1,10)
 501  FORMAT(X9A8,XA8)
 5    COLCOUNT=1
 3    CHARAC   =RDBYTE (BUFFER,COLCOUNT,8,64,MASK)
 7    COLCOUNT=COLCOUNT+1 $ CHARTYPE=RDBYTE  (TYPETAB,CHARAC+1,16,8,7)
      IF (CHARTYPE) 130,131
 131  CHARTYPE=8
 130  STAGE=RDBYTE (DECISION(STAGE),CHARTYPE,8,64,MASK)
      IF (STAGE-11) 10,9,100
 9    STAGE=10
 10   GO TO (11,12,11,11,15,16,11,18,19,20    ),STAGE
 12   IF (CHARAC-1R+    ) 21,11
 21   SIGN=1    $ GO TO 11
 15   IF OVERFLOW FAULT 22,22
 22   IF (CHARAC    -10) 24,23,25
 23   CHARAC    =0
 24   I=N+N       $ I=I+I+N$ I=I+I+CHARAC$ IF OVERFLOW FAULT 25,26
 26   N=I $ GO TO 11
 25   DIGITS=DIGITS+1 $ GO TO 11
 16   DIGITS=DIGITS-1 $ GO TO 15
 18   IF ( (CHARAC-1R+)*(CHARAC-1R )) 27,11
 27   ESIGN=1    $ GO TO 11
 19   IF (CHARAC    -10) 30,29
 29   CHARAC    =0
 30   EXPONENT  =EXPONENT*10+CHARAC     $ GO TO 11
 20   I=WORD.AND.7700000000000000B $ IF (I) 11,28
 28   WORD=64*WORD $ WORD=WORD.OR.CHARAC $ GO TO 11
C     SCAN COMPLETE. PREPARE FOR EXIT
 100  STAGE=STAGE+1-1RA        $GO TO(101,102,103,104,105,106,107), STAGE
 101  I=COLCOUNT-1 $ PRINT 502,I,         CHARAC     $ GO TO 1
 502  FORMAT(19H INPUT ERROR COLUMN I3,11H CHARACTER  02)
 103  COLCOUNT=COLCOUNT-1 $ CHARAC    =OLDCHAR
 102  ISIT= CHARAC* 1000000000000000B $ ISIT=7R        .OR.ISIT
 114  RETURN
 105  COLCOUNT=COLCOUNT-1
 104  DIGITS=DIGITS+(1-ESIGN-ESIGN)*EXPONENT $TESDC=1$N=N*(1-SIGN-SIGN)
 108  IF (FLOATING) 124,109
 109  ISIT=N $ IF (DIGITS) 126,114,111
 111  IF OVERFLOW FAULT 112,112
 112  DO 120 I=1,DIGITS $ N=ISIT+ISIT $ N=N+N+ISIT
 120  ISIT=N+N      $ IF OVERFLOW FAULT 113,114
 113  PRINT 503
 503  FORMAT(18H INTEGER TOO LARGE)
      ISIT=3777 7777 7777 77778 $ GO TO 114
 126  DIGITS=-DIGITS $ DO 125 I=1,DIGITS
 125  ISIT=ISIT/10 $ GO TO 114
 124  X=N $ K=XABSF(DIGITS) $ J1=K/100 $ K=K-100*J1
      J2=K/10 $ K=K-J2*10 $ Y=XPA(J1+1)*XPB(J2+1)*XPC(K+1)
      IF (DIGITS) 115,116,117
 115  X=X/Y
 116  ISIT=I $ GO TO 114
 117  X=X*Y $ GO TO 116
 107  COLCOUNT=COLCOUNT-1
 106  TESDC=-1 $ ISIT=WORD $I=WORD.AND.770000000000000B$ IF (I) 114,132
 132  WORD=WORD*64 $ WORD=WORD.OR.1R  $ GO TO 106
      ENTRY RDTYPE $ ISIT=TESDC $ GO TO 114
      ENTRY RDPRINT  $ PRINTING=ISIT $ GO TO 114
      ENTRY RDMODE $ GO TO (201,202),ISIT
 201  DECISION(10)=DECISION(11)$ GO TO 114
 202  DECISION(10)=DECISION(12)$ GO TO 114
      END
      FUNCTION RDBYTE (I,J,K,KK,MASK)
      DIMENSION I(10)
      N=(J-1)/K $ M=K+K*N-J $ II=I(N+1) $ IF (M) 1,1,2
 2    NN=1 $ DO 3 N=1,M
 3    NN=NN*KK $ II=II/NN
 1    RDBYTE =II.AND.MASK $ END

CARD TOTAL =   94
```

FIG. 1

All that remains to be done is to change J to floating point and multiply it by 1000.

The decision-matrix is stored within the routine RDNUM by the following FORTRAN-63 data statement:

DIMENSION DECISION (10)
DATA (DECISION = 8H532= =1BA, 8H54CCCCCA, ..., 8HGGG= =GGA)

That is, the matrix is represented by a table of BCD characters and the symbol 10, which has no BCD equivalent, is represented by =. Suppose we are in stage J and encounter a character of type K; then the new stage is specified by the kth character in the table entry DECISION(J). The table as it is given here would read the string ALPHA99 as word ALPHA, followed by number 99. We can change this interpretation by

$$\text{DECISION(10)} = \text{8H=GG= =GGA}$$

ALPHA99 will now be read as word ALPHA99. The instruction DECISION(10) = 8HGGG= =GGA would restore the original interpretation.

8. Conclusion

It is a well-known fact that data errors caused by human fallibility are responsible for wasting an appreciable amount of computer time. The standard FORTRAN input routine will sometimes take datum that "looks reasonable" and interpret it in a way that is reasonable but unfortunate. It is possible to design a routine which both is easy to use and powerful, yet which interprets reasonable-looking data in the desired way.

Acknowledgment. An IBM 704 routine which had some of the features described here was written by the author while working for the United Kingdom Atomic Energy Authority at Risley. This routine was not published as a separate item but was distributed as part of a larger program [5]. Mitchell [9] revised and added to the routine in preparing the 7090 FORTRAN version at Risley. These earlier versions were written in machine language (FAP) and did not use the decision tables method. The final CDC FORTRAN-63 version reported here and the work leading to it was supported by Contract No. AT GEN-10(11-1) Project Agreement 9.

RECEIVED MARCH, 1964

REFERENCES

1. BAILEY, M. J., BARNETT, M. P., AND FUTERELLE, R. P. Format-free input in FORTRAN. *Comm. ACM, 6*, 10 (Oct. 1963), 605-608.
2. BROOKER, R. A. Mercury autocode. *Ann. Rev. Autom. Programming, 1* (1960), 93.
3. BUMGARNER, L. L. The Oak Ridge ALGOL compiler for the CDC 1604—preliminary programmers manual. CDC List No. SPD-02, Control Data Corp., Minneapolis, 1963.
4. CDC FORTRAN-63 Reference Manual. CDC 1604/1604A Computer, CDC List No. 527, Control Data Corp., Minneapolis, 1963.
5. HASSITT, A. A computer program to solve the multigroup diffusion equations. TRG Report No. 229(R), UKAEA, Risley, 1962.

6. ——. Format free input using the FORTRAN list statement. SHARE Distribution No. 1473, 1963.
7. HEINBERG, G. M., AND GRESSET, G. L. An experiment in the automatic verification of programs. *Comm. ACM. 6*, 10 (Oct. 1963), 610.
8. IBM 7090/7094 Programming Systems. In *Fortran IV Language*, IBM List No. C28-6274-1, 1963.
9. MITCHELL, M. F. Flexible Decimal and Alphabetic Input Routine for FORTRAN II. SHARE Distribution No. 1469, 1963.
10. PYM, J., AND FINDLAY, G. K. The ELLIOT 803 autocode mark 2. *Ann. Rev. Autom. Programming, II* (1961), 77.
11. RAMSHAW, W. B., Input Routine. SHARE Distribution No. 1025, 1961. (An earlier version of this routine dates back to 1955.)
12. VASILAKOS, G. J. A decision matrix as the basis for a simple data input routine. *Comm. ACM, 5*, 12 (Dec. 1962), 599.
13. YARBOROUGH, L. D. Input data organization in FORTRAN. *Comm. ACM, 5*, 10 (Oct. 1962), 508.

Reducing Truncation Errors by Programming

JACK M. WOLFE
Brooklyn College, Brooklyn, New York

In accumulating a sum such as in a numerical integration with a large number of intervals, the sum itself becomes much larger than the individual addends. This may produce a less accurate sum as the number of intervals is increased.

Separate variables can be established as accumulators to hold partial sums within various distinct intervals. Thus, the extensive successive truncations are eliminated.

When accumulating a sum such as would be involved in performing a numerical integration with a large number of intervals, the sum itself becomes much larger than the individual addends. Consequently, because of the shifting to line up the decimal points, there is a loss of significant digits from the right-hand end of the addends. As the sum grows increasingly larger, the resulting truncation for the remaining addends becomes more extensive. Thus an increase in the number of intervals beyond a certain point produces results that are actually less accurate rather than more accurate.

By a simple programming technique this error can be reduced significantly. Separate variables can be established as accumulators to hold the sums that are in various intervals. For example, $S1$ can be used to hold sums from 1 to 9.9999999; $S2$ can be used to hold sums from 10 to 99.999999; $S3$ for sums from 100 to 999.99999; etc. The summing is done in the lowest level accumulator until it is about to overflow. At that point the sum is added to the next higher accumulator, and it is then initialized to the value of the number that was to have been added to it. The successive accumulators are treated similarly in chain fashion. By this technique the successive and progressively more serious truncations that result from the usual procedure of accumulating the sum are eliminated.

Figure 1 shows a portion of a flowchart that illustrates the principle of these special accumulators. The illustration refers to the area under the curve $Y = f(X)$ in the interval from $X = A$ to $X = B$, calculated by means of the trapezoid rule using N-intervals. In this illustration it is assumed that $.1 \leqq Y < 10$. Thus we shall use $S1$, the lowest level accumulator, to hold sums up to .99999999. $S2$ will then be used to hold sums from 1.0000000 to 9.9999999; $S3$ for sums from 10.000000 to 99.999999; $S4$ for sums from 100.00000 to 999.99999; etc. As many of these accumulators should be established as may be needed to hold the largest sum possible, which may often be estimated easily if one knows the largest number of intervals that he intends to employ in the calculations and an approximate value of the average Y or even of the maximum Y. In the flowchart shown in Figure 1 it is assumed for illustrative purposes that the total sum will not exceed 999.99999. To allow for a possibly larger total sum, $S5$ can be established to hold sums from 1000 to 9999.9999. The flowchart can be extended in similar fashion but its basic logic would be the same as in the illustration.

It should be noted that the stepping up of X, indicated in the flowchart at connector 5, must not be performed by simply adding H to X. For if H were very small compared with X, the same kind of truncation errors would appear in this sum as in the sum of the Y values. Thus if X were stepped up by use of the statement $X = X + H$, it would be necessary to set up a set of special accumulators for this summing in addition to the set of special accumulators for the summing of the Y values. These special accumulators can be avoided in the stepping up of X, however, by taking advantage of the fact that X is always being incremented by the same amount, H. Thus we may use the statements

$$X = A + C * H$$

$$C = C + 1$$

to step up X. The value of C would have been initialized as 2 prior to entering the main loop where Y is calculated.

As an illustrative application of these principles the area under the line $Y = .55555550$ in the interval from 0 to 1 was calculated using various numbers from 10 to 9000 as the number of intervals. In this calculation $S1$ was used to hold sums up to 9.9999999. Since the largest total sum is 9000 times the Y-value, the highest level accumulator must accommodate a number with four significant digits to the left of the decimal point. Thus the accumulators were set as follows: $S2$ for sums from 10.000000 to 99.999999; $S3$ for sums from 100.00000 to 999.99999; and $S4$ for sums from 1000.0000 to 9999.9999. The errors in the area found using these special accumulators are shown in Table 1 along with the corresponding errors found using the usual method of obtaining a sum by establishing a single variable to hold the total sum.