FORTRAN Programming in Nuclear Medicine

Till Noever

Emory University Hospital, Atlanta, Georgia

This is the second in a series of four Continuing Education articles on computers in nuclear medicine. After studying this article, the reader should be able to: 1) understand the basics of FORTRAN programming as they apply to nuclear medicine; and 2) discuss the meaning of key components in a FORTRAN program.

The programming language FORTRAN (FORmula TRANslator), in a variety of versions, has been a ubiquitous feature of the programming world for many years—and, by sheer inertia, is probably going to be around for an indefinite time, despite its detractors' hopes for its demise.

FORTRAN is a compiled language (*1*), which means that the user must write a source program using a text editor program. The source program is then processed through another program, called a "compiler," which translates the english-like text into machine language. The output from this is called an object or relocatable binary file, which must then be integrated by yet another program into the specific computer environment. All these procedures are fairly standard and straightforward once an individual gets used to them. A good explanation of the procedure has been previously published in the *Journal* (*1*) and should be considered prerequisite reading for understanding of the current article.

The compiler checks the source program for any errors detectable at that level (incorrect statements, nonsensical or incomplete iteration loops, inaccessible points in a program, etc.), and will request their correction before producing the object file.

Introductory texts on FORTRAN programming abound, and the reader should refer to these for more detail (*2-4*). It is assumed that the reader is familiar with such terms and concepts as "statement," "main program," "subroutine," "variable," "declaration," "buffer," etc., in relation to computer languages in general and FORTRAN in particular. For those who are not, a brief glossary of some of the key terms of the language, as well as some information on FORTRAN statements are given in Appendices F and G.

Although almost all implementations of the language differ to a lesser or greater degree, there is a large common core of statements that represent the "portable" section of FORTRAN. Portable means that a program written using these common-to-all FORTRAN statements may be, in their source code versions, transported between computers implementing FORTRAN and should recompile and run without any problems.

The nonportable sections of the language implementation are usually those statements relating to input and output—to terminals and files—or function calls closely related to operations specific to the implementation on a particular computer. In addition, some later versions of FORTRAN (such as FORTRAN 77) have been enhanced to include features such as conditional execution of blocks of code without the requirement of GOTO statements: a feature that was not implemented in the original versions of the language. Programs using these advanced features are unlikely to compile on most other machines.

There are other portability problems that are unrelated to system considerations. One such problem is that a few computers will actually compile FORTRAN programs written in lower case characters whereas the vast majority will not. Therefore, even if one happens to have one of the former systems, it is probably a good idea to continue writing FORTRAN statements in UPPER CASE characters only.

Portability is, by the way, not a purely academic issue. With the proliferation of inexpensive computer systems on the market, it becomes quite feasible and efficient to develop and pretest algorithms on a personal computer and then transport them to a clinical system for final testing. The proper coding and structuring of programs, in order to make as many of them as portable as possible, is therefore quite important.

Because of the ready availability of FORTRAN compilers, they may be found on almost all programmable nuclear medicine systems as the first-choice programming language. They usually incorporate a host of system supported "library" functions for display and data access.

Completion of a polished program is rewarding, but lacking specific programming details can be frustrating and is eased by using thoughtful programming structure. The following sections, with respect to FORTRAN programming in nuclear medicine applications, will concentrate on those aspects of programming that enhance: 1) a program's readability and portability; 2) a program's memory use, execution speed, and interactivity; and 3) the translation of programs in other languages (especially BASIC) into FORTRAN.

PROGRAMMING APPLICATIONS

Readability

Although many of the remarks in the following section apply

For reprints contact: Till Noever, Imaging Science Laboratory, Department of Radiology, Emory University Hospital, 1364 Clifton Road N.E., Atlanta, GA 30322.

to the use of any programming language, those that are FORTRAN-specific will be evident from the context. However, programs that are written for nuclear medicine application (or any other diagnostic or therapeutic computing application) are quite different from the small 10–100-line programs that are usually generated for individual use on a personal computer or from those that are perhaps used for a research project on the hospital's or university's computing facility. The difference lies in their use. In this instance, programs are meant to aid in the diagnosis of disease, and the integrity of their algorithms and function is therefore paramount. Any program, and be it ever so tentative, has the potential for expansion and eventual routine application. This implies that the initial designer/programmer may have to surrender his program to others for further development or possible clinical testing and validation. What may have been a tentative idea may become a clinically usable procedure, incorporating all the ethical and legal ramifications that it may entail. The keywords in the development of such programs are, therefore, "documentation" and "readability."

This means, most importantly, that the program must contain concise, to-the-point comments that describe the purpose of every main program or subroutine. Functional subsections (e.g., a program loop to filter the data in an image) should be preceded by a comment on their purpose. Such in-line comments need not be long, but should exist nevertheless. An individual who is reasonably adept with the programming language and the algorithms that are used ought to be able to follow the program flow in its source-coded version without major difficulty. Alterations to a given version of the program should also be appropriately commented, and former versions should be kept for reference. Experience shows that even apparently trivial or "cosmetic" changes to a working program may otherwise result in many frustrating hours of work in an attempt to trace an elusive "bug."

For those programmers familiar with BASIC, comments in a compiled program, unlike those in the universally implemented interpreted version of that language, neither add to the space of the loaded program nor do they slow it down. The compiler totally ignores comments when generating machine compatible codes. Their existence is confined solely to the source program.

FORTRAN's comment lines have to be preceded by the letter C in the first column, which will instruct the compiler to ignore everything in the current line. For full portability, any program line containing no code at all (e.g., lines that are used for visual separation of program segments) should also be prefixed by a C.

The programmer is advised to resist the temptation to add comments later. Experience shows that the most lucid and effective commenting takes place when the code is initially written and the programmer uses his comments to clarify his own train of thought. The extra time and effort spent will always pay off. It may even improve on the logic in the original program since what looks good on a flow chart may not appear as neat and readable when coded.

Portability

Buffer size. Program portability may be ensured by keeping buffer sizes as small as possible, which will then ensure that machines with smaller memory space will be able to accept a program or subroutine without major modifications in program logic. It should be noted that, in this context, a large number of nuclear medicine computers (assuming they are FORTRAN-programmable) still allow only ~ 64 kilobytes of effective programming space. Many programs written for these systems will allow, after inclusion of all the system utilities and library functions necessary to run the program, only an uncomfortably small portion of free space for program codes, buffers, and variable allocations. In such programs, if two 64 × 64 resolution image buffers (one for input/unprocessed data and one for output/processed data) are included, then 16 kilobytes (two images at 64 × 64 pixels per image that use 2 bytes per pixel) of memory are taken up before programming even starts. The same calculation for 128 × 128 resolution images shows that these images will use all of the 64 kilobytes available. Hence, the need for judicious memory use.

Separation procedures. Portability will also be ensured by separating processing, input/output program segments, or subroutines.

Since the system-specific calls for graphics, keyboard input/output and file read/write functions are most likely to differ between computer systems. Consequently, they should be kept apart and preferably be placed in subroutines which are called from the main processing program. Subroutines can then be rewritten for each different machine, keeping the remainder of the code unchanged.

Special caution should be applied with regards to the use of formatted keyboard input (e.g., the "ACCEPT" statement). An unacceptable formatted input at runtime causes the program to crash in some operating systems whereas other systems either catch the error and reprompt for input or else generate nonsensical input conversions that may cause puzzling results. If the time exists, a generalized input handler for keyboard data may be written in order to avoid such crashes. One such example, written in a portable version, is given in Appendix A. The level of commenting in this routine probably exceeds anything that can be realistically expected of a scientific programmer (although it is usually required in a commercial environment). This subroutine is almost crash-proof and provides decoding of the input string (which is seen as just a string of characters and terminated by a single carriage return) into integer and real numbers. It performs a function not unlike the BASIC "INPUT" statement. This subroutine cannot accept exponentially formatted inputs. It is, however, a good example on which to expand, and it will run under most FORTRAN versions.

Appendix B contains the code for another program. This program performs a two-dimensional 3 × 3 convolution filter on a 64 × 64 image that uses only one image buffer for input and output image and assigns the actual filtering operation to a 3 × 64 integer buffer and an intermediate output array, local to the subroutine, which take up only another 512, rather than 8,192, bytes of storage.

Appendix C illustrates an example of how to combine these two previous subroutines with some other completely fictitious functions which will, presumably, read and write out images into a program that allows two-dimensional filtering of 64 × 64 images with operator-specific convolution coefficients (*1*). Note that there is clear distinction between the use of system-specific utilities, keyboard input/output, and data processing. The layout of the program logic should make it comparatively easy to rewrite it for most systems.

MEMORY USE, SPEED, AND PROGRAM INTERACTIVITY

The subroutine in Appendix B is exemplary of a basic dilemma faced by almost all programmers: there is an inverse relationship between the amount of memory used for storage of data (input, intermediate, or output) and the length of the program required to process the data, in addition to the amount of time required for processing.

An example of how program size decreases when more memory space becomes available is illustrated in Appendix D. This is a reworked version of the filtering program (Appendix B), but this time the input and output buffers (while requiring an additional 7,808 bytes of storage for image data are saving maybe 100 bytes maximum in terms of program code) are separated. In this version, the speed will also increase dramatically because it has now become unnecessary to read data into the temporary buffer. If only a single image is to be processed, the difference in speed at execution time may not be immediately noticeable. If, however, the image is, for example, a multiframe dynamic study, then the delays will mount and eventually become quite intolerable (especially with more complicated and mathematically more demanding operations).

Another example of the ubiquitous compromise is shown in the programs in Appendix E. All these are intended to draw a ring, 10-pixels thick, on a fictitious display. In order to do this, it is necessary to calculate a series of 512 × 10 x-y coordinate pairs (total number of pairs = 10,240) that lie around a given center. Since we are dealing with rectangular coordinates, the generation of these points involves the calculation of sines and cosines and their placement onto the screen using a subroutine called WRTPTS (fairly representative of the type of graphics utilities supplied with programmable systems) that allows us to draw a specifiable number of points (given in x-y coordinate pairs) from a buffer on a screen.

Three possible versions of this program are given, and each contains a different way of making the compromise.

Version 1 calculates each pixel coordinate inside the loop and then outputs the pixel to the screen. This program occupies the smallest possible program space, and will, in addition, probably take anywhere up to one minute to run (assuming one pixel may be drawn with each new scan of the video signal across the screen). This is a ballpark figure which is, however, based on experience.

Version 2 takes account of the fact that sines and cosines are computationally complicated procedures, which should not really be placed into the innermost processing loop. It therefore calculates them only once, places the value of sines and cosines into two lookup tables, and then reduces the operations inside the loop to simple multiplications. This will save a lot of time, but it also requires an additional 512 (points) × 2 (sine and cosine) × 4 (4 bytes/floating point number) = 4,096 bytes in terms of storage.

Version 3 extends this theory still further by observing that execution of the subroutine WRTPTS always takes the same amount of time, and it would, therefore, be better not to write points singly, but in batches of 512. This will speed up execution dramatically, but it will only add another 512 (points) × 2 (points per coordinate) × 2 (bytes per coordinate) = 2,048 bytes to the storage space required.

Of course, intermediate solutions to the extremes presented here are possible, but they all will involve a careful consideration of the compromise between the possible and the desirable.

The possible is dictated by the machine and computing environment used. On the other hand, the desirable is defined by the fact that most processing in nuclear medicine involves a fairly high degree of operator-machine interaction, and slowly executing programs are not only annoying, but also inefficient, since they either tie up otherwise needed manpower or impose long additional working hours.

PROGRAM TRANSLATION

BASIC and FORTRAN

In all probability, many readers, if they have had any acquaintance with programming, will probably be familiar with one of the many implementations of the language BASIC because it is found on almost all personal and large system computers.

Many potentially useful programs for medical applications are created in the close interactive development environment BASIC provides. As these programs grow more complex (and slower, using the interpreted versions of BASIC) and require increased sophistication of specific display and database functions that are to be found only on the systems provided by medical computer manufacturers, the need to transport them will increase. BASIC will have to be translated into FORTRAN in some way.

Although some provisos must be followed if the transfer is to be accomplished without major rearrangement of program logic and layout, this is less complicated than it sounds. In order to do this, it will help if the BASIC program is modified (or maybe written initially) with the following in mind:

1. FORTRAN has virtually none of the convenient BASIC keyboard input/output and string handling functions. They would have to be specially written (e.g., the subroutine KEYIN in Appendix A, which might be used to replace INPUT), and the effort would probably be worth it. These functions can then be included in any programs requiring them. As far as the initial BASIC program is concerned, the best way of minimizing trans-

lation problems is to keep any such functions well separated from the processing sections of the program. Better still, do not use any such special functions at all if possible. Enter any variables into the interpreted BASIC program rather than inputting them at runtime.

2. Do not use more than one statement per program line! In BASIC, this tends to slow one down, but the nature of FORTRAN code format makes it necessary.
3. Any "IF-THEN-ELSE" statements in BASIC should be replaced with following the rules: (a) Avoid combinations involving "ELSE;" and (b) The only instruction following an "IF" should be a "GOTO" (or "THEN GOTO," depending on implementation).
4. All "FOR-NEXT" loops become "DO statement#-CONTINUE" type loops.
5. Data type declarations (INTEGER, REAL, DOUBLE PRECISION, STRING) should be kept as clear as possible to avoid type errors in translation. Explicit declarations should be used in the FORTRAN implementation (i.e., use INTEGER, REAL, DOUBLE, etc. declarators rather than DIMENSION in the FORTRAN version).
6. Place all BASIC subroutines in neat blocks near the end of your program. Recall that the "GOSUB line#-RETURN" statement becomes an explicit "CALL subroutine name (argument list, if any)" in the FORTRAN implementation. Write the BASIC program so that it sets up the variables required to run the "GOSUB" subroutine immediately before the GOSUB statement. This will allow easy identification of argument requirements in the FORTRAN implementation.
7. File access in BASIC (using the concept of "channels" and sequential/record-linked, and formatted/unformatted read-write access) is often not too dissimilar from that in FORTRAN. Syntactic differences notwithstanding, little trouble should be found here. It is advisable, however, to keep the program sections that perform these functions well distinct (and commented) from "processing" type operations.

Given the nature of the interpreted language and its tendency toward slow execution, most of these programming prescriptions may be generally useful because they might help in debugging the program as well.

Experience also shows that BASIC programs tend to grow "organically" with little forethought as to the program structure. Consequently, they are difficult to translate unless their logic is "decoded" by using devices like flowcharts or "pseudocode" (a shorthand english-type of program flow description with a syntactic structure not dissimilar to a structured programming language rather than FORTRAN or BASIC).

The approach that is used really does not matter in the end. The programmer will have to follow his/her own preference and judgement while keeping in mind the application requirements.

In conclusion, the aphorism that "practice makes perfect" is a truth that applies even more so to programming. It is also equally true that anybody reasonably familiar with BASIC programming will be able to master FORTRAN without great difficulty.

Given that a particular problem is understood (i.e., formulation of a description of the algorithm required to process a given set of data to generate some particular kind of result), it is a much smaller step from that point to writing a simple program to execute that procedure than most people think. The barrier between the person and his or her ability to program is mostly psychological, and there is little correlation between it and their general abilities, intelligence, or training. This is especially applicable to those individuals who work in an environment where some sort of scientific data evaluation takes place every day—any nuclear medicine department, for example.

The problem is then reduced to one of opportunity and motivation—both of which are, unfortunately, irreducible to algorithmic solution.

ACKNOWLEDGMENTS

The author gratefully acknowledges the support and assistance rendered by Joel A. Oates, CNMT, in the review and preparation of this article.

REFERENCES

1. Erickson JJ. Nuclear medicine computers—Software. *J Nucl Med Technol* 1985;13:140–49.
2. McCracken DD. *A Guide to FORTRAN Programming.* New York: John Wiley, 1961.
3. Murrel PW, Smith CL. *An Introduction to FORTRAN IV Programming.* New York: Int. Textbook Co., 1970.
4. Nickerson RC. *Fundamentals of FORTRAN 77 Programming.* Boston: Little, Brown & Co., 1985.

APPENDIX A
Example Keyboard Input Routine

```
C*************************************************************************************************
C
C     PROGRAM TO GET A STRING OF ASCII CHARACTERS FROM THE KEYBOARD AND
C     DE-CODE THEM INTO INTEGER AND FLOATING-POINT NUMBER, IF POSSIBLE
C
      SUBROUTINE KEYIN ( IPROMPT, ISTRING, NUMIN, INUM, FNUM, IER )
C
C     ARGUMENTS:
C                 IPROMPT    —    PROMPT STRING (UP TO 80 CHARACTERS,
C                                 TERMINATED BY A NULL BYTE)
```

```
C               ISTRING     —    STRING OF CHARACTERS (UP TO 80 LONG)
C                                FROM THE KEYBOARD. CHARACTERS WILL BE
C                                "BYTE-PACKED" INTO THE ARRAY UPON EXIT
C                                FROM THIS ROUTINE. THIS ARRAY IS ALSO
C                                USED TO HOLD THE INITIAL INPUT DATA.
C               NUMIN       —    NUMBER OF INPUT CHARACTERS IN INPUT
C                                STRING
C               INUM        —    INPUT STRING DECODED AS INTEGER (IF
C                                POSSIBLE)—ELSE SET TO 0
C               FNUM        —    INPUT STRING DECODED AS SINGLE-PRECISION
C                                FLOATING POINT NUMBER (IF POSSIBLE—
C                                OTHERWISE THIS WILL BE SET TO 0.0 !)
C               IER         —    ERROR RETURN. SHOULD BE = 1 IF EVERYTHING
C                                WENT O.K.
C
C     SUBROUTINES/LIBRARY FUNCTIONS USED:
C     (DEPENDING ON COMPILER AND SYSTEM THESE MAY HAVE TO BE
C     DECLARED AS "EXTERNAL")
C     GCHAR    —    GET CHARACTER FROM KEYBOARD
C     PCHAR    —    OUTPUT CHARACTER TO VDU
C     BYTE     —    FORTRAN BYTE FUNCTION USED TO EXTRACT A BYTE INTO
C                   LOWER BYTE OF ONE-WORD INTEGER
C
C
C     COMMENTS:  ONLY PRINTABLE CHARACTERS AND THE RUBOUT/DELETE
C                CHARACTER ARE ALLOWED AS INPUTS.
C                THE "RETURN" CHARACTER (ASCII 13) TERMINATES THE
C                STRING.
C
C                PRINTABLE CHARACTERS START AT ASCII VALUE 32 (BLANK)
C                AND END AT ASCII VALUE 126 (~). RUBOUT/DELETE IS
C                ASSUMED TO BE ASCII VALUE 127.
C
C                THE "BACKSPACE" CHARACTER IS ASSUMED TO BE ASCII 8.
C
C     ORIGINAL VERSION CREATED BY name, date
C
C     UPDATE RECORD:
C
C
C***********************************************************************************************
C
C     D E C L A R A T I O N S
C
C     SUBROUTINE ARGUMENTS:       (DECLARE ALL BY TYPE AND AVOID
C                                 "DIMENSION" DECLARATOR)
      INTEGER IPROMPT(80),ISTRING(80),NUMIN,INUM
      REAL      FNUM
C
C-----------------------------------------------------------------------------------------------
C
C     BEGINNING OF EXECUTABLE CODE
C
C     DISPLAY PROMPT STRING
      DO 1 I=1,80
C     TERMINATOR?
      IF(BYTE(IPROMPT,I).EQ.0)GOTO 5
      CALL PCHAR(BYTE(IPROMPT,I),IER)
C     OUTPUT ERROR?
      IF(IER.NE.1)GOTO 1000
1     CONTINUE
5     CONTINUE
C
C
C     INITIALIZE VARIABLES AND STRING BUFFER
      NUMIN=0
      INUM=0
      FNUM=0.0
```

```
C
      DO 10 I=1,80
      ISTRING(I)=0
10    CONTINUE
C
C     SOME ADDITIONAL LOCAL VARIABLES THAT HAVE TO BE INITIALIZED
      VAL=0.
      POWER=1.
C
C     GET INPUT STRING— MAXIMUM OF 80 CHARACTERS
C     CRITERION FOR 80 CHARACTERS IS THAT THE VALUE OF "NUMIN" HAS TO
C     BE GREATER OR EQUAL TO 80
C
C
C     LOOP INDEX IS BIG TO ALLOW FOR MORE THAN 80 CHARACTERS
C     SINCE WE COUNT DELETE/RUBOUT AS WELL!
      DO 100 I=1,000
C
C     GET A CHARACTER
C     PLACE INTO ICHAR. IER IS ERROR RETURN, IF NOT EQUAL TO ONE THEN
C     EXIT FROM THIS ROUTINE FLAGGING THE ERROR
20    CALL GCHAR( ICHAR, IER )
      IF(IER.NE.1)GOTO 1000
C     SEE IF WE HAVE TERMINATION CHARACTER
      IF(ICHAR.EQ.13)GOTO 110
C     OR A RUBOUT/DELETE
      IF(ICHAR.EQ.127)GOTO 50
C     ELSE SEE IF IT IS A PRINTABLE CHARACTER— AND IF NOT THEN GET
C     ANOTHER INPUT
      IF(ICHAR.LT.32.OR.ICHAR.GT.126)GOTO 20
C
C     PRINTABLE CHARACTER WAS INPUT. NOW STORE IT IN THE STRING BUFFER,
C     INCREMENT INPUT CHARACTER COUNTER AND ECHO CHARACTER ON SCREEN
      ISTRING(I)=ICHAR
      NUMIN=NUMIN+1
      CALL PCHAR(ICHAR, IER)
C     IF AN ERROR HERE THEN EXIT
      IF(IER.NE.1)GOTO 1000
C     ELSE GET ANOTHER CHARACTER
C     IF WE ARE STILL ALLOWED MORE
      GOTO 90
C
C
C     HANDLE RUBOUT/DELETE
C
C     DELETE PREVIOUS CHARACTER (IF THERE WAS ONE!)
50    IF(NUMIN.EQ.0)GOTO 20
C
C     BACKSPACE ONE CHARACTER
      CALL PCHAR(8,IER)
      IF(IER.NE.1)GOTO 1000
C     WRITE A BLANK OVER CHARACTER
      CALL PCHAR(32,IER)
      IF(IER.NE.1)GOTO 1000
C     BACKSPACE AGAIN
      CALL PCHAR(8,IER)
      IF(IER.NE.1)GOTO 1000
C     AND DECREMENT INPUT CHARACTER COUNTER
      NUMIN=NUMIN-1
C     MAKE SURE THIS IS NEVER LESS THAN ZERO
      IF(NUMIN.LT.0)NUMIN=0
C     GET ANOTHER CHARACTER
      GOTO 20
C
90    CONTINUE
C     FULL BUFFER?
      IF(NUMIN.GE.80)GOTO 110
C
```

```
C     LOOP BACK TO BEGINNING FOR ANOTHER INPUT
100   CONTINUE
C
C     THIS IS WHERE WE GET TO AFTER INPUT STRING HAS BEEN TERMINATED
110   CONTINUE
C
C     NOW DECODE AS A REAL NUMBER (AND DO INTEGER TRUNCATION LATER)
C
C     EMPTY STRING?— IF SO THEN JUST EXIT
      IF(NUMIN.EQ.0)GOTO 900
C
C     ELSE DECODE
C
C     CHECK IF POSITIVE OR NEGATIVE
C     (NCHAR COUNTS CHARACTERS DECODED, I POINTS AT INPUT BUFFER, SIGN
C     FLAGS SIGN
C
200   NCHAR=0
      I=1
      SIGN=1.
C     "+" OR "-" ?
      IF(ISTRING(I).NE.43.AND.ISTRING(I).NE.45)GOTO 210
C     IS SO THEN SET SIGN, ELSE PROCEED
      NCHAR=NCHAR+1
      IF(ISTRING(I).EQ.45)SIGN=-1.
      I=2
C
210   CONTINUE
      VAL=0.
C     BETWEEN "0" AND "9" ?
220   IF(ISTRING(I).LT.48.OR.ISTRING(I).GT.57)GOTO 300
C
C     ITEMP HOLDS VALUE DECODED ASCII
      ITEMP=ISTRING(I)-48
      VAL=10.*VAL+FLOAT(ITEMP)
      I=I+1
C     TRY ANOTHER CHARACTER FOR DECODE
      GOTO 220
C
C     DECIMAL POINT?
300   CONTINUE
C
      IF(ISTRING(I).NE.46)GOTO 400
C
C     ELSE GET DECIMAL PART
C     (PROCEED SIMILARLY AS ABOVE)
      NCHAR=NCHAR+1
      I=I+1
C
350   CONTINUE
      IF(ISTRING(I).LT.48.OR.ISTRING(I).GT.57)GOTO 400
      ITEMP=ISTRING(I)-48
      VAL=10*VAL+FLOAT(ITEMP)
      POWER=10*POWER
      I=I+1
      GOTO 350
C
C     SEE IF O.K.
400   CONTINUE
C     STRING MUST BE TERMINATED BY 0
      IF(ISTRING(I).NE.0)GOTO 900
C     ONLY + OR - OR . ?
      IF(NCHAR.GE.I)GOTO 900
C
C     ELSE MAKE DECIMAL THE VALUE OF THE REAL NUMBER
      FNUM=SIGN*VAL/POWER
C
C     TRY TO MAKE AN INTEGER (BUT ONLY BETWEEN 32767 AND -32767, ELSE
C     SET THIS TO ZERO!)
C
C     INTEGER WIL BE   R O U N D E D !
      INUM=INT(FNUM+.5)
      IF(FNUM.GT.32767..OR.FNUM.LT.-32767.)INUM=0
C
C     AND NOW PACK THE STRING
900   CONTINUE
C     EMPTY STRING?
      IF(NUMIN.LE.0)GOTO 1000
C
C     PACK IT!— IBPTR POINTS AT OUTPUT BYTE — FUNCTION "BYTE" IS USED
C     TO PACK DATA
C
      IBPTR=1
      DO 950 I=1,NUMIN
      BYTE(ISTRING,IBPTR)=BYTE(ISTRING,I*2)
      IBPTR=IBPTR+1
950   CONTINUE
C
C     AND ZERO OUT THE REMAINDER
      DO 960 I=IBPTR,160
      BYTE(ISTRING,I)=0
960   CONTINUE
C
C     E X I T   FROM THIS PROGRAM
1000  CONTINUE
C
      RETURN
C
C
C     INDICATE PROGRAM END TO COMPILER
C
      END
C
C
```

APPENDIX B
Program for Two-Dimensional Filtering on a 64 × 64 Image

```
C***********************************************************************************************
C
C     PERFORM 3x3 CONVOLUTION ON A 64x64 IMAGE
C
      SUBROUTINE SFILT( IDATA, FILKER)
C
C     ARGUMENTS:
C                   IDATA    —    64x64 INPUT/OUTPUT ARRAY (INTEGER)
C                   FILKER   —    3x3 ARRAY CONTAINING FILTERING KERNEL
C
C     COMMENTS:  3x3 CONVOLUTION WILL BE PERFORMED ON ALL PIXELS EXCEPT
C                PERIPHERAL ONES. THE LATTER WILL BE SET TO THE VALUE OF
C                THE PIXEL CLOSEST TO THEM.
C                OUTPUT DATA WILL OVERWRITE INPUT DATA!
C
C***********************************************************************************************
C
C     DECLARATIONS:
      INTEGER IDATA(64,64)
      REAL FILKER(3,3), FKSUM, SUM
C
C     LOCAL VARIABLES:
C     ITMP WILL HOLD LINES TO BE PROCESSED, IOUT WILL HOLD OUTPUT LINE
      INTEGER ITMP(64,3),IOUT(64)
C
C     EXECUTABLE CODE
C
```

```
C     SUM KERNEL VALUES
      FKSUM=0.
      DO 20 I=1,3
          DO 10 J=1,3
              FKSUM=FKSUM+FILKER(I,J)
10        CONTINUE
20    CONTINUE
C
C
C     LINES 2 TO 63 (COUNTED BY 'I')
C
      DO 200 I=2,63
C
C         READ DATA INTO ITMP
          DO 120 K=1,3
              DO 110 L=1,64
              ITMP(L,K)=IDATA(L,I-2+K)
110           CONTINUE
120       CONTINUE
C
C         WRITE OLD IOUT BACK INTO IDATA
          DO 125 J=1,64
              IDATA(J,I-1)=IOUT(J)
125       CONTINUE
C
C         PROCESS DATA IN ITMP
          DO 150 J=2,63
              SUM  =0.
              DO 140 K=1,3
                  DO 130 L=1,3
                     SUM=SUM+ITMP(J-2+L,K)*FILKER(L,K)
130               CONTINUE
140           CONTINUE
C             ROUND THE RESULT
              IOUT(J)=INT(SUM/FKSUM  +  0.5)
150       CONTINUE
C         DO FIRST AND LAST COLUMN
          IOUT(1)=IOUT(2)
          IOUT(64)=IOUT(63)
C
200   CONTINUE
C
C     NOW SET FIRST AND LAST LINES
      DO 250 I=1,64
         IDATA(I,1)=IDATA(I,2)
         IDATA(I,64)=IDATA(I,63)
250   CONTINUE
C
C     DONE
      RETURN
C
      END
```

APPENDIX C
Simulated Main Program for Performing Image Filtering

```
C***************************************************************************************************
C
C     PROGRAM IMGFILT
C
C     PURPOSE: TO PERFORM 3x3 CONVOLUTION ON 64x64 IMAGE
C
C     SUBROUTINES USED:
C     GETIMAGE    —    GET IMAGE FROM NUCLEAR DATABASE
C     PUTIMAGE    —    PUT IMAGE INTO NUCLEAR DATABASE
C     SFILT       —    PERFORM 3x3 CONVOLUTION FILTER
C     KEYIN       —    KEYBOARD INPUT HANDLER
C
```

```
C     COMMENTS:
C     NAME OF INPUT AND OUTPUT IMAGES, AS WELL AS THE
C     VALUES OF THE CONVOLUTION KERNEL WILL BE INPUT BY OPERATOR
C
C     THIS ROUTINE WILL ASSUME THAT INPUT IMAGES HAVE ONLY ONE FRAME.
C     EXTENSION TO MULTIPLE FRAME IMAGES ARE OBVIOUS.
C
C*****************************************************************************************
C
C     DECLARATIONS:
C
C     IMAGE DATABASE NAMES
      INTEGER IPNAME(10),OPNAME(10)
C
C     ARRAYS TO HOLD IMAGES
      INTEGER IMAGE(64,64)
C
C     KEYBOARD BUFFERS
      INTEGER IPROMPT(40),ISTRING(40),NUMIN,INUM
      REAL FNUM
C
C     FILTER KERNEL
      REAL FILKER(9)
C
C     EXECUTABLE CODE
C
C     GET IMAGE FROM DATABASE
10    CONTINUE
      CALL KEYIN("ENTER INPUT IMAGE NAME:",IPNAME,NUMIN,INUM,FNUM,IER)
C     ERROR— TRY AGAIN . . .
      IF(IER,NE.1)GOTO 10
C     NULL ENTRY— ABORT PROGRAM
      IF(NUMIN.EQ.0)GOTO 1000
C     ELSE WE HAVE A NAME AND SO TRY AND GET THE IMAGE FROM DATABASE
      CALL GETIMAGE(IPNAME,IMAGE,IER)
C     ERROR (I.E. IER.NE.1 !)— TRY AGAIN
      IF(IER.EQ.1)GOTO 20
      TYPE "ERROR—", IER, "WHEN ATTEMPTING TO READ IMAGE FROM DATABASE"
      GOTO 10
C
20    CONTINUE
C     ENTER THE FILTER COEFFICIENTS
      TYPE "ENTER THE FILTER COEFFICIENTS IN THE FOLLOWING ORDER:"
      TYPE "#1 = (1,1)     #2 = (1,2)     #3 = (1,3)"
      TYPE "#4 = (2,1)     #5 = (2,2)     #6 = (2,3)"
      TYPE "#7 = (3,1)     #8 = (3,2)     #9 = (3,3)"
C
C-------INPUT LOOK
      DO 50 I=1,9
C
30    CONTINUE
      TYPE "#",I
      CALL KEYIN("ENTER COEFFICIENT:",ISTRING,NUMIN,INUM,FNUM,IER)
C     TRY AGAIN ON ERROR
      IF(IER.NE.1)GOTO 30
C     ABORT IF NOTHING ENTERED FIRST TIME AROUND, ELSE START AGAIN
      IF(NUMIN.NE.0)GOTO 40
C
      IF(I.EQ.1)GOTO 1000
      GOTO 20
C
40    CONTINUE
      FILKER(I)=FNUM
C
50    CONTINUE
C-------END OF INPUT LOOP
C
C     DO THE FILTER
      CALL SFILT(IMAGE,FILKER)
```

```
C
C     AND PUT IMAGE BACK INTO THE DATABASE
60    CONTINUE
      CALL KEYIN("ENTER OUTPUT IMAGE NAME:",OPNAME,NUMIN,INUM,FNUM,IER)
C     ERROR— TRY AGAIN...
      IF(IER.NE.1)GOTO 60
C     NULL ENTRY— RE-DO FROM BEGINNING
      IF(NUMIN.EQ.0)GOTO 10
C     ELSE WE HAVE A NAME AND SO TRY AND GET THE IMAGE FROM DATABASE
      CALL PUTIMAGE(OPNAME,IMAGE,IER)
C     ERROR (I.E. IER.NE.1 !)— TRY AGAIN
      IF(IER.EQ.1)GOTO 70
      TYPE "ERROR—", IER, "WHEN ATTEMPTING TO WRITE IMAGE TO DATABASE"
      GOTO 60
C
70    CONTINUE
C
C     COMPLETED
1000  STOP
      END
```

APPENDIX D
Program for Two-Dimensional Filtering on a 64 × 64 Image

```
C*****************************************************************************************************
C
C     PERFORM 3x3 CONVOLUTION ON A 64x64 IMAGE
C
      SUBROUTINE SFILT( IDATAIN, IDATAOUT, FILKER)
C
C     ARGUMENTS:
C                 IDATAIN     —    64x64 INPUT ARRAY (INTEGER)
C                 IDATAOUT    —    64x64 OUTPUT ARRAY (INTEGER)
C                 FILKER      —    3x3 ARRAY CONTAINING FILTERING KERNEL
C
C     COMMENTS:  3x3 CONVOLUTION WILL BE PERFORMED ON ALL PIXELS EXCEPT
C                PERIPHERAL ONES. THE LATTER WILL BE SET TO THE VALUE OF
C                THE PIXEL CLOSEST TO THEM.
C                INPUT AND OUTPUT DATA ARE IN SEPARATE ARRAYS!
C
C*****************************************************************************************************
C
C     DECLARATIONS:
      INTEGER IDATAIN(64,64),IDATAOUT(64,64)
      REAL FILKER(3,3), SUM, FKSUM
C
C     EXECUTABLE CODE
C
C     SUM KERNEL VALUES
      FKSUM=0.
      DO 20 I=1,3
         DO 10 J=1,3
            FKSUM=FKSUM+FILKER(I,J)
10       CONTINUE
20    CONTINUE
C
C     I COUNTS LINES, J COLUMNS
      DO 200 I=2,63
         DO 190 J=2,63
            SUM=0
            DO 140 K=1,3
               DO 130 L=1,3
                  SUM=SUM+ITMP(J-2+L,I-2+K)*FILKER(L,K)
130            CONTINUE
140         CONTINUE
C        ROUND THE RESULT
         IDATAOUT(J,I)=INT(SUM/FKSUM + 0.5)
190      CONTINUE
```

```
C          DO FIRST AND LAST COLUMN
           IDATAOUT(1,I)=IDATAOUT(2,I)
           IDATAOUT(64,I)=IDATAOUT(63,I)
200    CONTINUE
C
C      NOW SET FIRST AND LAST LINES
       DO 250 I=1,64
           IDATAOUT(I,1)=IDATAOUT(I,2)
           IDATAOUT(I,64)=IDATAOUT(I,63)
250    CONTINUE
C
C      DONE
       RETURN
C
       END
```

APPENDIX E
Drawing Rings on a Graphics Overlay on the Display

Version 1

This method occupies the least space and takes the longest to execute.

```
C***********************************************************************************************
C
       SUBROUTINE RING (IXCENTER,IYCENTER,IRADIUS)
C
C      PURPOSE:
C      DRAW A RING 10 PIXELS THICK ON THE DISPLAY (DISPLAY HAS 512x512
C      PIXEL RESOLUTION).
C
C      ARGUMENTS:
C      IXCENTER    —    X-COORDINATE OF CENTER OF RING ON DISPLAY
C      IYCENTER    —    Y-COORDINATE OF CENTER OF RING ON DISPLAY
C      IRADIUS     —    OUTER RADIUS OF RING
C
C      NOTE:   COORDINATES ARE ASSUMED TO START AT LOWER LEFT HAND CORNER
C      OF DISPLAY AT COORDINATE (1,1)
C
C      SUBROUTINES USED:
C      WRTPTS      —    WRITE PIXEL COORDINATES TO SCREEN
C      (READS "N" POINT PAIRS FROM A BUFFER AND INTERPRETS THEM AS X-Y
C      COORDINATE PAIRS)
C      FORTRAN SIN AND COS FUNCTIONS (ACCEPTING RADIANS AS ARGUMENTS)
C
C***********************************************************************************************
C
C      LOCAL BUFFERS
       INTEGER ICOORDS(2)
C
C      DRAW RING DIRECTLY:
C
C      RADIANS PER POINT
       RADPP=3.1412/256.
C
C      10 PIXELS THICK
       DO 100 I=1,10
           TMP1=FLOAT(IRADIUS-I)*RADPP
C          512 PIXELS ON CIRCUMFERENCE
           DO 50 J=1,512
               TMP2=FLOAT(J)*RADPP
C             X-COORDINATE
              ICOORDS(1)=XCENTER+TMP1*COS(TMP2)
C              Y-COORDINATE
               ICOORDS(2)=YCENTER+TMP2*SIN(TMP2)
               CALL WRPTS(1,ICOORDS,IER)
50         CONTINUE
100    CONTINUE
C
```

```
C     FINISH
      RETURN
C
      END
```

Version 2

This method occupies ~ 2,100 bytes more than Version 1, but it will run noticeably faster. The main body of the program only is included here.

```
C****************************************************************************************************
C
C     LOCAL BUFFERS
      INTEGER ICOORDS(2)
      REAL SINES (512),COSINES(512)
C
C     RADIANS PER POINT
      RADPP=3.1412/256.
C
C     CALCULATE SINE AND COSINE LOOKUP TABLE
      DO 10 I=1,512
      TMP2=FLOAT(I)*RADPP
      SINES(I)=SIN(TMP2)
      COSINES(I)=COS(TMP2)
10    CONTINUE
C
C     CALCULATE CIRCLES
C     10 PIXELS THICK
      DO 100 I=1,10
         TMP1=FLOAT(IRADIUS-I)*RADPP
C        512 PIXELS ON CIRCUMFERENCE
         DO 50 J=1,512
C           X-COORDINATE
            ICOORDS(1)=XCENTER+TMP1*COSINES(J)
C           Y-COORDINATE
            ICOORDS(2)=YCENTER+TMP1*SINES(J)
            CALL WRPTS(1,ICOORDS,IER)
50       CONTINUE
100   CONTINUE
C
C     FINISH
      RETURN
C
      END
```

Version 3

This method occupies ~ 4,200 bytes more than Version 1, but it will run much faster than either of the preceding methods. The main body of the program only is included here.

```
C****************************************************************************************************
C
C     LOCAL BUFFERS
      INTEGER ICOORDS(1024)
      REAL SINES(512),COSINES(512)
C
C     RADIANS PER POINT
      RADPP=3.1412/256.
C
C     CALCULATE SINE AND COSINE LOOKUP TABLE
      DO 10 I=1,512
      TMP2=FLOAT(I)*RADPP
      SINES(I)=SIN(TMP2)
      COSINES(I)=COS(TMP2)
10    CONTINUE
```

```
C     CALCULATE CIRCLES
C     10 PIXELS THICK
      DO 100 I=1,10
         TMP1=FLOAT(IRADIUS-I)*RADPP
C        512 PIXELS ON CIRCUMFERENCE
C        CALCULATE A FULL RING BEFORE WRITING IT OUT
         DO 50 J=1,512
C           X-COORDINATE
            ICOORDS((J-1)*2+1)=XCENTER+TMP1*COSINES(J)
C           Y-COORDINATE
            ICOORDS(J*2)=YCENTER+TMP1*SINES(J)
50       CONTINUE
         CALL WRPTS(512,ICOORDS,IER)
100   CONTINUE
C
C     FINISH
      RETURN
C
      END
```

APPENDIX F
Glossary of Programming Terminology

Buffer:	(also called array) an area of storage for more than one number, logical entity, or string that is reserved by the program.
Compiler:	a program which translates the source code into machine language, and produces an "object" or "relocatable binary" file that represents the next stage towards generating a running program.
Conditional execution:	execution of a single or a block of statements that depend on the outcome of a logical test on numerical, logical, or string variables.
Constant:	a symbolic or explicit representation in a program of a number, logical entity, or string whose content is fixed and may not be changed by the program during execution.
Declaration:	a statement about the type of data to be stored in a variable or array for which storage space is to be allocated in a program.
Linker:	(also called "loader") a program that takes as its input the object file generated by the compiler as well as a range of required system functions (stored in files called "system libraries") and produces a program that can then be run on a given computer.
Loop:	a series of statements executed repeatedly that are subject to the varying conditions imposed on them by the program.
Source code:	(also called "source program") the program entered by the programmer in such "high-level" languages as FORTRAN, Pascal or C.
Statement:	syntactically valid instruction in the source code of a FORTRAN program (see Appendix G).
Statement block:	a group of one or more statements that are logically considered to be a single unit and are delineated by language-specific markers to indicate the beginning and end of a block.
Variable:	a symbolic representation in a program of a number, logical entity, or string that may have a varying content as the program executes.

APPENDIX G
FORTRAN Statements*

The first five columns of any FORTRAN statement may be used only as follows:

1. Column 1 may contain the letter C, in which case the compiler will ignore everything else in this line.
2. Columns 1–5 may contain a statement number (integer) to be used as a target for GOTO statements or conclusion of "DO"-loops.
3. Column 6 may contain either a blank or a character. If the latter is the case, the current statement line is considered to be a continuation of the previous statement line.
4. Columns 7 to end-of-line (usually not more than 80 columns) contain the FORTRAN statement for this line (or some comment, if there is a C in Column 1 of this line).

Note that FORTRAN ignores blanks (the "SPACE" character in the statement section of a FORTRAN program line).

Statement	Meaning
GOTO line number	Transfer execution to line with number indicated in statement.
IF (condition) statement	Execute "statement" if "condition" is true.
CALL subr(arg list)	Execute subroutine "subr" with the arguments in "arg list."
DO 1n count= start, end, step	Execute the block of statements between this statement and the statement with line-number "1n" (inclusive) subject to the following conditions: Set the loop counter "count" to the value "start" on first entry. Increment loop counter by "step" upon each iteration of the loop. Exit from loop if "count" is either greater than "ending value" (if "step" is greater than 0), or if "count" is less than "end" (if "step" is less than 0).
CONTINUE	Proceed to the logically following statement. This statement is used mainly with a preceeding statement number, serving as a target for transfer of program execution. In some compilers the ends of DO-loops must be indicated by "CONTINUE."
SUBROUTINE name (list)	Indicates that the following program is a subroutine of "name" with the arguments in "list."
DIMENSION, INTEGER, REAL, LOGICAL, DOUBLE, COMPLEX	Examples of type declarations when reserving storage space for variables or arrays in a program. Note that FORTRAN compilers will design default types to arrays when "DIMENSION" is used. These types will be determined by the first letter in the name of the variable or array.

*The precise meaning of the terms above may vary from system to system.

FORTRAN PROGRAMMING IN NUCLEAR MEDICINE

For each of the following questions select the best answer. Then circle the number on the reader service card that corresponds to the answer you have selected. Keep a record of your responses so that you can compare them with the correct answers, which will be published in the next issue of the *Journal.*

A. *FORTRAN is:*
142. a basic language.
143. written directly in machine language.
144. a compiled language.
145. all of the above.

B. *A "portable" FORTRAN program:*
146. cannot be recompiled.
147. is written using common-to-all FORTRAN statements.
148. is written in advanced code statements.
149. all of the above.

C. *The compiler:*
150. checks the source program for any errors detectable at that level.
151. translates the English-type text into machine language.
152. will request the error correction before producing the object file.
153. all of the above.

D. *Program comments:*
154. should be concise statements at the end of each subroutine.
155. should be an in-line statement of purpose before each main program or subroutine.
156. need not be included in updated program versions.
157. all of the above.

E. *FORTRAN comment lines:*
158. do not add to the space of the loaded program.
159. are ignored by the compiler if preceeded by the letter "C" in the first column.
160. do not slow down the program in any way.
161. all of the above.

F. *The output of the compiler is:*
162. a source code file.
163. an object file.
164. a current statement file.
165. none of the above.

G. *Most nuclear medicine compilers will have a host of system-supported __________ for display and database access.*
166. compilers.
167. "library" functions.
168. object files.
169. source files.

H. *The keywords in the development of clinical programs are:*
170. documentation and portability.
171. readability and portability.
172. documentation and readability.
173. readability and memory use.

I. *"Cosmetic" changes to a working program may in fact alter the program's overall function, causing a "bug."*
174. true.
175. false.

J. *Any line in a FORTRAN program preceded by a letter "C" is an indication that ____________.*
176. the line should be executed twice.
177. the line should be ignored.
178. there is a subroutine call at that line.
179. the line is complete and ready for use.

K. *FORTRAN compilers may be found on ________ programmable nuclear medicine systems.*
180. very few.
181. most.
182. all.

L. *FORTRAN has virtually __________ of the convenient BASIC keyboard input/output and string handling functions.*
183. none.
184. all.
185. some.

Your answers to the above questions should be returned on a reader service card (found in the back of the *Journal*) no later than September 1, 1986. Remember to supply your name and address in the space provided on the card; also, write your VOICE number following your name. Your VOICE number appears on the upper left hand corner of your *Journal* mailing label. No credit can be recorded without it. A 70% correct response rate is required to receive 0.1 CEU credit for this article. Members participating in this continuing education activity will receive documentation on their VOICE transcript, which is issued in March of each year. Nonmembers may request verification of their participation but do not receive transcripts.