INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION

INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
34100 TRIESTE (ITALY) - P.O.B. 586 - MIRAMARE - STRADA COSTIERA 11 - TELEPHONE: 2240-1
CABLE: CENTRATOM - TELEX 460392-I

SECOND SCHOOL ON ADVANCED TECHNIQUES
IN COMPUTATIONAL PHYSICS
(18 January - 12 February 1988)

SMR.282/ 3

FORTRAN 8X - THE EMERGING STANDARD

M.METCALF
CERN, European Lab. for Particle Physics, Geneva, Switzerland

CERN-Data Handling Division
DD/87/1
M. Metcalf
January 1987
Revised

Fortran 8x - The Emerging Standard

ABSTRACT

Since 1979, the Fortran standardization committee, X3J3, has been labouring over a draft for the next version of the standard. Its initial intention of publishing this draft in 1982 was hopelessly optimistic, and at best it may be ready this year. However, a number of fundamental issues have been thrown up over the past two years, such that it is clear that unanimity cannot be achieved: efficiency versus functionality; safety versus obsolescence; small and simple or big and powerful?

This paper reviews the current state and content of Fortran 8x, and attempts to bring out some of the controversial issues surrounding its development.

Presented at the
Conference on Computing in High-Energy Physics,
Asilomar, February 1987

FORTRAN is not a flower, but a weed. It is hardy, occasionally blooms, and grows in every computer.
A. Perlis

1. LANGUAGE EVOLUTION

The hardiness of the Fortran 'weed' is now an accepted, if remarkable, aspect of scientific and numerical data processing. Various reasons, or excuses, for its vigour have been put forward -- the magnitude of the existing investment, its ease of use, and the efficiency of its implementations -- but one which is sometimes forgotten is the fact that the language itself has evolved since it was first introduced in 1957. This oversight is most evident in the writings of its detractors, who often unfairly compare their own favourite language with code written in abysmal style in Fortran 66, conveniently ignoring the success of Fortran 77 which, given some self-imposed discipline [1], can be employed to write programs of a very high standard (compare the code in Figs. 1 and 2).

This evolution has its formal side. The language has twice been standardized in the framework of ANSI and ISO, in 1966 and 1978. Following the publication of the second standard, the technical committee responsible for the work, X3J3, re-formed to begin work on a third standard, with the intention of having it ready in 1982. This time scale was hopelessly optimistic, and even now at the time of writing, February 1987, the best estimate is that the draft will be published this year, with a final standard in 1989, just allowing the new standard to cheat a little and change its working name from Fortran 8x to Fortran 88.

What are the justifications for continuing to revise the definition of the Fortran language? One is to modernize it in response to the developments in language design which have been exploited in new languages, particularly PASCAL and ADA. Here X3J3 has the advantage of hindsight, and can avoid the pitfalls of adding trendy features like the already outmoded DO...WHILE, whilst drawing on the obvious benefits of concepts like data hiding. In the same vein is the need to eliminate the dangers of storage association, to abolish the rigidity of the outdated source form, and to improve further on the regularity and portability of the language.

However, taken together, these would constitute only a ragbag of items perhaps better obtained by switching to other languages. The real strength of the new standard will be its incorporation of powerful array processing features and of derived-data types, allowing users to access vector processor hardware using a convenient notation, and to define and manipulate objects of their own design. Unfortunately for those of us who would like

to see a new standard published by the end of this decade, there are still many obstacles to be overcome. Formally, X3J3 is currently at milestone eight of the eighteen milestones which have to be passed before a new standard receives full international recognition, and there are powerful voices stating that the advances it has made have gone too far, and that there should be some further retrenchment before it is acceptable to them. A ballot of X3J3 held in April 1986, whose result was 16 to 19 against the draft of that date, has already resulted in some slimming down, including the loss of BIT data type, in an attempt to reach a compromise between radicals who want a large range of new features and conservatives with more modest goals and who emphasize the importance of long-term backwards compatibility. Features which have been removed in this process will be listed in a special Appendix to the published draft standard. A second ballot whose result has just been announced, 29 to 7, shows that better agreement has been reached, but that no true consensus has been achieved, given that two of the dissenting votes are those of IBM and DEC claiming to act as representatives of their users.

The dissension within the committee can be regarded as healthy as long as it leads to a constructive debate on its objectives resulting in a final resolution of the difficulties. However, some of the issues are indeed difficult, with complete reconciliation unobtainable:

- Should Fortran 8x be innovative, or merely standardise existing practice?
- Should the language be small and simple, or big and powerful? Is the present proposal too big, making it impossible to fit into the small machines of 1990, impossible to be implemented by small software houses, and impossible to be understood by non-professionals (the bulk of Fortran' users)?
- Are users prepared to drop existing features, given 10 to 20 years' notice, or must all existing code work for ever?
- Are subsets of the language useful, or an impediment to portability?
- Is a core plus modules architecture really viable?
- Do users want a safe language, or one which permits them to write the tricky programs more often associated with assembly languages?
- Should the standard document be written for users, or for compiler writers, or for both (as at present)?
- Are the existing proposals difficult and inefficient to implement? Does that matter if users thereby have an easier life?
- Will the heap storage mechanism necessary for dynamic array allocation impair efficiency, as happened with PL/1?

- Will the presence of new features cause existing ones to be implemented less efficiently?

The development of the present document has not followed a straight path, and some items (for instance BIT data type and the significance of blanks) have been in and out several times. This has meant that the public has had some difficulty in following the committee's work, as informative articles such as Refs. [2] and [3] quickly became out of date, and even misleading. This present paper is a further attempt to give a feel for Fortran 8x, written at a time when some convergence can be detected, and users are pressing for a new standard quickly. Fuller details must await the publication of the draft of the standard for public comment, and of accompanying explanations as planned in Ref. [4].

2. BACKWARDS COMPATIBILITY

The procedures under which X3J3 works require that a period of notice be given before any existing feature is removed from the language. This means, in practice, a minimum period of one revision cycle, which for Fortran means a decade or so. The need to remove features is evident: if the only action of the Standards Committee is to add new features, the language will become grotesquely large, with many overlapping and redundant items. The solution finally adopted by X3J3 is to publish as an Appendix to the standard a set of three lists showing which items have been removed or are candidates for eventual removal.

The first list contains the Deleted Features, those which in the previous standard were listed as Obsolescent Features, and have now been removed. The list of Deleted Features is empty for Fortran 8x which contains the whole of Fortran 77.

The second list contains the Obsolescent Features, those considered to be redundant and little used, and which should be removed in the next revision (although that is not binding on a future committee). The Obsolescent Features are

Arithmetic-IF	Real and double precision DO-variables
Shared DO termination	DO termination not on CONTINUE (or ENDDO)
Alternate RETURN	Branch to ENDIF from outside block
PAUSE	ASSIGN and assigned GOTO

The third list contains the Deprecated Features, those features which become redundant with Fortran 8x, but whose heavy present usage requires them first to pass into the Obsolescent Features list at the next revision,

before final removal the time after that. They fall into two groups, those linked to storage association:

Assumed-size dummy arrays
Passing of an array element or character substring to a dummy array
BLOCK DATA
COMMON
ENTRY
EQUIVALENCE

and those which are made redundant by newer features:

Fixed source form
Specific names for intrinsic functions
Statement functions
Computed GOTO
Old form of the DATA statement
DOUBLE PRECISION
DIMENSION
*len character length specifier

In the remainder of this paper it should become evident why these features are no longer needed to write Fortran programs, the main body of the language, the so-called core, containing all that is required.

3. MAIN NEW FEATURES OF CORE FORTRAN 8x

3.1 Source form

The new source form allows free form source input, without regard to columns. Comments may be in-line, following an exclamation mark(!), and lines which are to be continued bear a trailing ampersand (&). The character set is extended to include the full ASCII set, including lower-case letters (which in Fortran syntax are interpreted as upper case.) The underscore character is accepted as part of a name, which may contain up to 31 characters. Thus, one may write a snippet of code such as

```
SUBROUTINE CROSS_PRODUCT (x, y, z) ! Z = X*Y
:
z1 = x(2) * y(3) -                    &
     x(3) * y(2)
```


3.2 Alternative style of relational operators

In response to user demand, the deliberate redundancy of an alternative set of relational operators is introduced. They are

```
< for .LT.        > for .GT.
<= for .LE.       >= for .GE.
== for .EQ.       <> for .NE.
```

enabling statements such as

```
IF (x < y .AND. z**2 >= radius_squared) THEN
```

to be written.

3.3 Specification of variables

The existing form of type declaration, as shown by

```
REAL a(5,5), b(5,5)
```

is extended to allow all the attributes of the variables concerned to be declared in a single statement. For instance, the statement

```
REAL, ARRAY(25), PARAMETER :: a = [25*0.], b = [25*1.]
```

declares the two objects *a* and *b* to have the attributes of being arrays of named constants, whose values are specified by the array constructors (see Subsection 3.9) following the equals signs. Many other attributes may also be specified, in particular real and complex variables may be specified to have a defined minimum precision and/or exponent range, as shown in

```
COMPLEX (PRECISION = 12,  EXPONENT_RANGE = 100), ARRAY(10) &
                                :: current
```

where the variable *current* is specified to have a minimum precision of 12 decimal digits and an exponent range of at least $10^{\pm 100}$. This facility is of great benefit in writing portable numerical software. Where constants corresponding to a given range and precision are required, a corresponding exponent letter must be chosen, for example

```
EXPONENT LETTER (12,100) C
:
current (3) = 15.6C57
```

For those who wish to define explicitly the type of all variables, the statement

```
IMPLICIT NONE
```

turns off the usual implicit typing rules.

3.4 CASE construct

The CASE construct allows the execution of one block of code, selected from several, depending on the value of an integer, logical, or character expression. An example is

```
SELECT CASE(3*i-j)
    CASE(0)      ! for 0
    :            ! executable code
    CASE(2,4:8)  ! for 2, 4 to 8
    :            ! executable code
    CASE DEFAULT! for all other values
    :            ! executable code
END SELECT
```

The default clause is optional; one of the clauses *must* be executed.

3.5 Loop construct

A new loop construct is introduced whose syntax is combined with that of the old form of the DO-loop. In a simplified form it is

```
[name:] DO [(control)]
        block of statements
        END DO [name]
```

(where square brackets indicate optional items). The control parameter, if omitted, implies an endless loop; if present, it may have one of two forms:

```
i = intexp1, intexp2 [, intexp3]
```


or

```
intexp4 TIMES
```

The optional name may be used in conjunction with CYCLE and EXIT statements to specify which loop in a set of nested loops is to begin a new iteration or which is to be terminated, respectively.

3.6 Program units

An enhanced form of the call to a procedure allows *keyword* and *optional* arguments, with *intent* attributes. For instance, a subroutine beginning

```
SUBROUTINE solve (a, b, n)
OPTIONAL, INTENT (IN) :: b
```

might be called as

```
CALL solve (n = i, a = x)
```

where two arguments are specified in keyword (rather than positional) form, and the third, which may not be redefined within the scope of the subroutine, is not given in this call. The mechanism underlying this form of call requires an *interface block* with the relevant argument information to be specified.

Procedures may be specified to be recursive, as in

```
RECURSIVE FUNCTION factorial(x)
```

The old form of the statement function is generalized to an *internal procedure* which allows more than one statement of code, permits variables to be shared with the host procedure, and contains a mechanism for overloading operators and assignment for derived-data types. This we shall return to in Subsection 3.14.

3.7 Extensions to CHARACTER data type

A number of extensions to the existing CHARACTER data type permit the use of strings of zero length:

```
a = ''
```

and the assignment of overlapping substrings:

```
a(:5) = a(3:7)
```

and introduce new intrinsic functions, such as TRIM, to remove the trailing blanks in a string. Some intrinsics, including INDEX, may operate, optionally, in the reverse sense.

3.8 Input/Output

The work in the area of input/output has been mainly on extensions to support the new data types, an increase in the number of attributes which an OPEN statement may specify, for instance to position a file or to specify the actions allowed on it, and to add a NAMELIST feature, illustrated by

```
NAMELIST/list/a, i, x
READ(unit, NML = list)
```

which would expect an input record of the form

```
&LIST   X = 4.3,   A = 1.E20,   I = -4
```

3.9 Array processing

The introduction of array processing features is one of the most important new aspects of the language. The reasons are threefold: an array notation defined in the language simplifies the syntax; new features extend the power of the language to manipulate arrays; and the concise syntax makes the presence of array processing obvious to compilers which, especially on vector processors, are able to optimize object code better.

An array is defined to have a *shape* given by its number of dimensions, or *rank*, and the *extent* of each one. Two arrays are conformable if they have the same shape. The operations, assignments and intrinsic functions are extended to apply to whole arrays on an element-by-element basis, provided that when more than one array is involved they are all conformable. When one of the variables involved is a scalar rather than an array, its value is distributed as necessary. Thus we may write


```
REAL, ARRAY(5, 20) :: x, y
REAL, ARRAY(-2:2, 20) :: z
:
z = 4.0 * y * SQRT(x)
```

In this example we may wish to include a protection against an attempt to extract a negative square root. This facility is provided by the WHERE construct:

```
WHERE (x>=0.)
   z = 4.0 * y * SQRT(x)
ELSEWHERE
   z = 0.
END WHERE
```

which tests x on an element-by-element basis.

A means is provided to select sections through arrays. Such sections are themselves array-valued objects, and may thus be used wherever an array may be used, in particular as an actual argument in a procedure call. Array sections are selected using a triplet notation. For an array

```
REAL, ARRAY(-4:0, 7) :: a
```

a(-3,:) selects the whole of the second row, and a(0:-4:-2, 1:7:2) selects in reverse order every second element of every second column.

Just as variables may be array-valued, so may constants. It is possible to define a rank-one array-valued constant as in

```
[1, 1, 2, 3, 5, 8]
```

and to reshape it to any desired form:

```
REAL, ARRAY(2, 3) :: a
a = RESHAPE([2, 3], [1 : 6])
```

where the first argument to the intrinsic function defines the shape of the result, and the second defines an array of the first six natural numbers.

3.10 Dynamic storage

Fortran 8x provides four separate mechanisms for accessing storage dynamically. The first is the IDENTIFY statement which is a dynamic alias permitting a section of an array, including a skew section such as a diagonal, to be referenced as an object with its own name. An example is

```
REAL, ARRAY(:), ALIAS :: diag ! type, rank, name
REAL, ARRAY(10, 10)   :: x
:
IDENTIFY (diag(i) = x (i, i), i = 1:10)
```

Following execution of this statement, the array *diag* may be referenced as usual, as in

```
diag (4) = q
```

or

```
CALL sub(diag (2:9))
```

The second mechanism is via the ALLOCATE and DEALLOCATE statements which, as their names imply, are used to obtain and return the actual storage required for an array whose type, rank, name, and allocatable attribute have been previously declared in the procedure:

```
REAL, ARRAY(:), ALLOCATABLE :: x
:
ALLOCATE(x(n:m)) ! n and m are integer expressions
:
x(j) = q
CALL sub(x)
:
DEALLOCATE (x)
```

Deallocation occurs by default, unless the array has the SAVE attribute, whenever a RETURN or END statement in the same procedure is executed. The fact that allocation and deallocation occur in random order implies an underlying heap storage mechanism.

The third mechanism, useful for local arrays with variable dimensions, is the *automatic* array:


```
SUBROUTINE sub(i, j, k)
REAL, ARRAY(i, j, k) :: x ! bounds from dummy arguments
```

whose actual storage space is provided (on a stack) when the procedure is called.

Finally, we have the *assumed-shape* array, whose storage is defined in a calling procedure, and for which only a type, rank, and name are supplied:

```
SUBROUTINE sub(a)
REAL, ARRAY(:, :, :) :: a
```

Various enquiry functions may be used to determine the actual bounds of the array:

```
DO (i = ELBOUND (a, 1), EUBOUND (a,1))
  DO (j = ELBOUND (a, 2), EUBOUND (a, 2))
   DO (k = ELBOUND (a, 3), EUBOUND (a, 3))
```

where ELBOUND and EUBOUND give the effective lower and upper bounds of a specified dimension, respectively. (The effective range may differ from the declared range because of an intervening change in the range caused by execution of a RANGE statement, which is not further described here.)

3.11 Intrinsic procedures

Fortran 8x defines about 100 intrinsic procedures. Many of these are intended for use in conjunction with arrays for the purposes of reduction (e.g. SUM), inquiry (e.g. RANK), construction (e.g. SPREAD), manipulation (e.g. TRANSPOSE), and location (e.g. MAXLOC). Others allow the attributes of the working environment to be determined (e.g. the smallest and largest positive real and integer values), and access to the system and real-time clocks is provided. A random number subroutine provides a portable interface to a machine-dependent sequence, and a transfer function allows the contents of a defined area of physical storage to be transferred to another area without type conversion occurring.

3.13 Derived-data types

Fortran has hitherto lacked the possibility of building user-defined data types. This will be possible in Fortran 8x, using a syntax illustrated by the example

```
TYPE staff_member
     CHARACTER(LEN=20)::first_name, last_name
     INTEGER::id, department
END TYPE
```

which defines a structure which may be used to describe an employee in a company. An aggregate can be defined as

```
TYPE(staff_member), ARRAY(1000)::staff
```

defining 1000 such structures to represent the whole staff. Individual staff members may be referenced as, for example, staff(no), and a given field of a structure as staff(no)%first_name, for the first name of a particular staff member. More elaborate data types may be constructed using the ability to nest definitions as in

```
TYPE company
     CHARACTER(LEN=20)::name
     TYPE(staff_member), ARRAY(1000)::staff
END TYPE
:
TYPE(company), ARRAY(20)::companies
```

to build a structure to define companies.

3.14 Data abstraction

It is possible to define a derived-data type, and operations on that data type may be defined in an internal procedure. These two features may be combined into a module which can be propagated through a whole program to provide a new level of data abstraction. As an example we may take an extension to Fortran's intrinsic CHARACTER data type whose definition is of a fixed and pre-determined length. A user-defined derived-data type, on the other hand, may define a set of modules to provide the functionality of a variable length character type, which we shall call *string*. The module for the type definition might be


```
MODULE string_type
     TYPE string(maxlen)
          INTEGER::length
          CHARACTER(LEN=maxlen)::string_data
     END TYPE String
END MODULE String_type
```

With

```
USE string_type
:
TYPE(string(60)), ARRAY(10)::cord
```

we define an array of 10 elements of maximum length 60. An actual element can be set by

```
cord(3) = 'ABCD'
```

but this implies a re-definition, or *overloading*, of the assignment operator to define correctly both fields of the element. This can be achieved by the internal procedure

```
SUBROUTINE c_to_s_assign(s,c) ASSIGNMENT
     TYPE (string(*))   ::s
          CHARACTER(LEN=*)::c
          s%string_data = c
          s%length      = LEN(c)
END SUBROUTINE c_to_s_assign
```

which can be included in the module, together with other valid functions such as concatenation, length extraction, etc., to allow the user-defined string-data type to be imported into any program unit where it may be required, in a uniformly consistant fashion.

4. CONCLUSION

This paper has tried to set out the background to the current revision of the Fortran language standard, and to give an overview of its principal new features. An idea of what a section of scalar code might look like is given in Fig. 3, a conversion of the code of Fig. 2. The difference for

array code would be more dramatic, as the total number of lines reduces by a factor of up to eight according to some initial tests.

The final question is whether the proposed standard, on balance, is what the high-energy physics community requires for its main-line data processing needs to the end of the century. Any standard is a compromise between conflicting points of view, as indicated in the first section. On the other hand, the presence of a widely accepted and implemented standard provides the only practical means to write and to use portable programs, and the evolution of that standard is the only means to escape from the temptation to use vendors' extensions or private dialects and languages (e.g. MORTRAN and LRLTRAN). It is my opinion that although the current proposal does not contain everything which we would like to see, particularly BIT data type and pointers, and although it contains some items of more limited appeal to our community, nevertheless its final acceptance and implementation are the only way to avoid a future of anarchy and chaos in the long-term.

REFERENCES

[1] Metcalf M., Effective FORTRAN 77 (Oxford Univ. Press, Oxford, 1985).

[2] Metcalf M., FORTRAN Optimization (Academic Press, London and New York, 1982 and 1985), Chapter 12.

[3] Metcalf M., Has FORTRAN a Future? Comp. Phys. Comm. 38 (1985), 199-210.

[4] Metcalf M. and Reid J., Fortran 8x Explained (Oxford Univ. Press, Oxford, 1987).

Figure legends

Fig. 1 A program using FORTRAN 66 features and style. Note the large number of statement labels.

Fig. 2 The same program as in Fig. 1 converted to use FORTRAN 77 features. Note the use of: a program name, lower case letters, CHARACTER data type, terminal oriented I/O with list-directed format and error recovery, IF...THEN...ELSE constructs, in-line format specifications, and expressions in output lists. Very few statement labels remain.

Fig. 3 The same program as in Fig. 2 converted to use some Fortran 8x features. Note the use of: in-line commentary, long names, the DO-construct, PROMPT on READ, alternative relational operators, the continuation mark &, and " as a delimiter. Only one statement label remains.

```
C       SOLUTION OF QUADRATIC EQUATION
      COMPLEX COMP(2)
    1 READ ( 5, 51) ANAME, N
   51 FORMAT(A6,I2)
      WRITE ( 6, 52) ANAME
   52 FORMAT(1H1,33HROOTS OF QUADRATIC EQUATIONS FROM A6)
      DO 21 I = 1, N
         READ ( 5, 53) A, B, C
   53    FORMAT(3F10.2)
         WRITE ( 6, 54) I, A, B, C
   54    FORMAT(1H0,8HSET NO. I2/5H A = F8.2,12X,4HB = F8.2,12X,4HC =
     +   F8.2)
         IF (A.NE.0.) GO TO 10
         IF (B.NE.0.) GO TO 7
         WRITE ( 6, 59)
   59    FORMAT(9H NO ROOTS)
         GO TO 21
    7    RLIN = -C/B
         WRITE ( 6, 55) RLIN
   55    FORMAT(7H LINEAR,25X,4HX = F10.3)
         GO TO 21
   10    D = B**2 - 4.*A*C
         IF (D.GE.0.) GO TO 17
         COMP(1) = CMPLX(-B/(2.*A), SQRT(-D)/(2.*A))
         COMP(2) = CONJG(COMP(1))
         WRITE ( 6, 56) COMP
   56    FORMAT(8H COMPLEX,21X,7HR(X1)= F10.3,11X,7HI(X1)= F10.3,/1H ,
     +   28X,7HR(X2)= F10.3,11X,7HI(X2)= F10.3)
         GO TO 21
   17    SQRTD = SQRT(D)
         REAL1 = (-B + SQRTD)/(2.*A)
         REAL2 = (-B - SQRTD)/(2.*A)
         WRITE ( 6, 57) REAL1, REAL2
   57    FORMAT(6H REAL 25X,5HX1 = F10.3,13X,5HX2 = F10.3)
   21 CONTINUE
      WRITE ( 6, 58) ANAME
   58 FORMAT(8HOEND OF A6)
      GO TO 1
      END
```

Fig. 1

```
      PROGRAM QROOTS
*
*     Solution of quadratic equation
      CHARACTER*6 ANAME
      COMPLEX COMP
*
*     Read title and number of solutions from terminal
    1 READ ( *, *, END= 999) ANAME, N
      WRITE ( *, '('' ROOTS OF QUADRATIC EQUATIONS FROM '',A)') ANAME
      DO 21 I = 1, N
         READ ( *, *) A, B, C
         WRITE ( *,  '(A,I2/3(A,F8.2:TR12))')
     +               '0SET NO. ', I, ' A = ', A, 'B = ', B, 'C = ', C
         IF (A.EQ.0.) THEN
            IF (B.NE.0.) THEN
               WRITE ( *, '('' LINEAR'',TR25,A,F10.3)') 'X = ', -C/B
            ELSE
               WRITE ( *, '('' NO ROOTS'')')
            ENDIF
         ELSE
            D = B**2 - 4.*A*C
            IF (D.LT.0.) THEN
               COMP = CMPLX(-B/(2.*A), SQRT(-D)/(2.*A))
               WRITE ( *, '('' COMPLEX'',TR21,''R(X1)= '',F10.3,TR11,
     +                    ''I(X1)= '',F10.3/T30,''R(X2)= '',F10.3,TR11,
     +                    ''I(X2)= '',F10.3)') COMP, CONJG(COMP)
            ELSE
               SQRTD = SQRT(D)
               REAL1 = (-B + SQRTD)/(2.*A)
               REAL2 = (-B - SQRTD)/(2.*A)
               WRITE ( *, '('' REAL '',TR25,2(A,F10.3:TR13))')
     +                    'X1 = ', REAL1, 'X2 = ', REAL2
            ENDIF
         ENDIF
   21 CONTINUE
      WRITE ( *, '(''0END OF '',A)') ANAME
      GO TO 1
  999 END
```

Fig. 2

```
PROGRAM QROOTS        ! Solution of quadratic equation
!
CHARACTER (LEN=6) :: ANY_NAME
COMPLEX COMP
!
DO
   READ ( *, *, PROMPT = 'Name, no. of equations', END= 999) ANY_NAME, N
   WRITE ( *, '(" Roots of quadratic equations from ",A)') ANY_NAME
   DO ( N TIMES)
      READ ( *, *, PROMPT = 'Coefficients of equation') A, B, C
      WRITE ( *,  '(A,I2/3(A,F8.2:TR12))')                                  &
                  '0Set no. ', I, ' A = ', A, 'B = ', B, 'C = ', C
      IF (A == 0.) THEN
            IF (B <> 0.) THEN
               WRITE ( *, '(" Linear",TR25,A,F10.3)') 'X = ', -C/B
            ELSE
               WRITE ( *, '(" No roots!")')
            ENDIF
      ELSE
         D = B**2 - 4.*A*C
         IF (D < 0.) THEN
            COMP = CMPLX(-B/(2.*A), SQRT(-D)/(2.*A))
            WRITE ( *, '(" Complex",TR21,"R(X1)= ",F10.3,                   &
                TR11,"I(X1)= ",F10.3/T30,"R(X2)= ",F10.3,                   &
                TR11,"I(X2)= ",F10.3)') COMP, CONJG(COMP)
         ELSE
            SQRTD = SQRT(D)
            REAL1 = (-B + SQRTD)/(2.*A)
            REAL2 = (-B - SQRTD)/(2.*A)
            WRITE ( *, '(" Real ",TR25,2(A,F10.3:TR13))')                   &
                       'X1 = ', REAL1, 'X2 = ', REAL2
         ENDIF
      ENDIF
   END DO
   WRITE ( *, '("0End of ",A)') ANY_NAME
END DO
999 END
```

Fig. 3