A Dynamic (FORTRAN) programming system

by JULIUS A. ARCHIBALD, JR.
State University College
Plattsburgh, New York

ABSTRACT

In recent years, new insights into the nature of programming languages have been obtained from the comparative study of natural and programming languages. These studies reveal that programming languages are deficient in their ability to adapt both to new requirements and new means for communicating thoughts. A means of alleviating these difficulties, through a dynamic, structured expansion of an established programming language (FORTRAN) is provided.

INTRODUCTION

During recent years, there have been some rather significant advancements in man's understanding of programming languages. Some ten or so years ago, and continuing into the present, we find a continuing growth of interest in the concepts of structured programming.[1,2] More recently, there has been the suggestion that the nature of programming languages can be better understood by the study of natural languages and the drawing of analogies between these two different types of languages.[3] One particularly useful analogy has been suggested, that between English, a poor, but useful natural language, and FORTRAN, a poor, but useful, programming language.[4] In doing this, we note that one of the major differences between these media of thought is their adaptability to changing requirements. Specifically, we note, both through a study of literature, and a review of our own usage of the language, that English has readily adapted both to the needs of expressing new thoughts, and to the needs of better ways of expressing and communicating old thoughts. English has thus evolved in a timely manner. Indeed, the development of English has been concurrent with, rather than lagging behind, the development of human knowledge. We also note, regretfully, that programming languages in general, and FORTRAN in particular do not share this characteristic. Programming languages as we now know them, are incapable of dynamic development or evolvement. Programming methods have changed, and languages have failed to keep pace.

The static nature of programming languages is a problem of particular concern today. The earliest programming languages, such as FORTRAN, were, of necessity, created before our present day understandings of the nature and structure of algorithmic processes and programming languages were attained. (Versions of FORTRAN were in use some ten years before the notion of structure was developed.) The time has come for us to benefit from the new understandings of the structure of algorithmic processes and programming languages developed in the last ten years within the framework of FORTRAN. The ideal solution would be to extend FORTRAN so as to include the new structures.

There are two problems. First of all, FORTRAN has been standardized.[5,6] On the positive side, this was good for purposes of definition. We now have a precise understanding as to exactly what is FORTRAN and what is not FORTRAN. Even more important, this understanding is the same in Boston, Atlanta, Los Angeles, and Seattle. The definition is not subject to local dialects. On the negative side, however, along with standardization came stagnation. The existence of a standard has successfully stifled the initiative needed to make the language flexible, and adaptive. To further complicate matters, those who were not content with the status quo mandated by the standard have ventured off into their own private extensions in such a manner as to produce a set of mutually incompatible super-languages of the original common language. The overall effect has not been beneficial to the development of programming languages.

The second problem is, in reality, not a problem of the programming languages themselves, but rather a problem resulting from differences in the use of natural and programming languages. The constructs of English become meaningful as a result of interpretation by thought processes resident in the human brain. The constructs of FORTRAN become meaningful as a result of interpretation by compilers (or interpreters) resident in a computer's memory. Thus, the increased flexibility (or adaptability) of natural languages over programming languages is not a question of the relative merits of the languages themselves, but rather

a matter of the superior ability of the human brain, as compared to a compiler, to program itself (or be programmed) to interpret new constructs. Thus, in order to attain for a programming language the flexibility and adaptability of a natural language, we must consider not only the language itself, but also the compiler (at least in general terms), as they are known to exist for the language. We thus must act upon Wirth's conclusion that "Language design is compiler construction."[7]

We refer here to a programming system on a given machine as being made up of a specific language and the compiler or interpreter used to interpret that language on the machine of concern. We will also refer to FORTRAN programming systems on a machine independent basis in the same manner that FORTRAN as a language is regarded on a machine independent basis. In the remainder of this paper, we will be working on obtaining flexibility and adaptability not merely for the language itself, but rather for the overall programming system. The objective is a dynamic, flexible extension of FORTRAN, as previously defined.[5] It will be accomplished through modification of the FORTRAN programming system.

It is noted in passing that the present use of static structured extensions of FORTRAN, which must be converted to standard FORTRAN, through the use of preprocessors, falls short of the above stated objective. They provide structure without flexibility. The need for a dynamic language has already been noted in the literature.[8] Conventional pre-processors simply are not dynamic. A second problem is that the preprocessors are not always machine independent. The various extensions themselves are not all consistent.

The approach of limiting oneself to a subset of the existing, standard FORTRAN from which things resembling structures can be formed is also inappropriate. This approach involves the further limitation of an already inadequate medium, rather than an extension of the medium to meet new challenges. Said in other words, this approach also fails to provide flexibility.

There is one more remaining question: "Why FORTRAN?" Again, the analogy with English. Users of English (both native and adopted) love to sit around and complain about how poor a language it is, how bad the grammar is, and how impossible spelling is. No one, however, has taken any serious steps to eliminate English. The best effort, to date, was the invention of a contrived artificial language, Esperanto,[3] a language with lots of merit and no potential. The problem is that too many people have already done, and are continuing to be doing, too many things with English to make a change feasible. Consider, for a moment, the problems that some of us are already experiencing from the change of just one small part of the language, namely the system of measures, to metric units. The same is true of FORTRAN, too many people have already done, and are continuing to do, too many things in FORTRAN to make a change feasible. FORTRAN is where the action is. The language, PL/I, intended to be a partial remedy, has no more potential than Esperanto. We are, as a matter of fact, suffering from addiction to FORTRAN. In the present situation, the pain of continued use is less than the pain of withdrawal. We thus support previous conclusions of others[3] as well as ourselves.[9]

IMPLEMENTATION

It has been shown that all programs, regardless of the language in which they are written, can be composed of three fundamental structures.[1] These structures consist of a sequential or composite structure, some variant of a predicate structure, and some variant of a repetitive structure. Regardless of the choice of variant, the result, that these structures are sufficient, remains valid.[10] (We do include, in our implementation, a fourth structure, the case structure, fully realizing that its use is not essential.) Thus, as will be seen, we will limit ourselves to D, D′, and BJn structures.[11] The point is that each structure is made up of certain, fixed parts, which, because they are independent of the language concerned, are referred to by psycholinguists as being "linguistic universals." [12]

The method of implementation that we use is to divide each program into two divisions. The first division consists of a definition of the form to be used for each part of each of the fundamental structures (or variants of the structures) to be used within the program. (We note that there may be several different definitions for each of the fundamental structures within the programs.) These structure definitions are followed by the second division, consisting of the source program itself, written in standard FORTRAN augmented by the just defined structures. Each programmer is free to define the fundamental structure in any manner that suits his (or her) convenience, thereby providing flexibility. Division one of the compilation will be the conversion of the user defined structures within the source program into standard FORTRAN statements, and the insertion of the resulting statements into appropriate places of the original source. In most situations, we "very strongly convert" in the sense of Ledgard and Marcotty, the various D, D′, and BJn structures to standard FORTRAN structures.[11] This conversion is followed by a routine FORTRAN compilation of the entire converted program. Any construct not part of standard FORTRAN must have been defined and converted. A program without structure definitions is assumed to be in standard FORTRAN, and, as such, does not require a division one of the compilation.

This differs from pre-processing, as it already exists today, in that each programmer, on a dynamic basis, creates his own form or forms for the fundamental structures. We reiterate that several different forms

of each structure may be used in each program. There are no pre-established forms for the structures themselves, only the parts of the structures remain invariant. This is the feature that provides the flexibility absent in the more conventional pre-processing.

All terminology used in describing structures will be compatible with FORTRAN usage. Thus, if in defining a predicate structure, the condition itself is referred to as being a logical expression, then the condition will require no further definition. Rather, it shall be assumed that all of the attributes and characteristics of FORTRAN logical expressions will apply.

This compatibility with FORTRAN will also apply to the structure definition statements, e.g., such statements will begin in Column 7, etc.

As a specific of the implementation, the actual conversion of the user defined structures into FORTRAN must itself be done in FORTRAN, and the resulting conversion must result in standard FORTRAN statements.[5] (This in itself is difficult because of the incomplete manner in which strings are defined in FORTRAN.) The conversion of user defined structures will require the insertion of new statements labels, and may, as well, require new variable names. In those cases where the existing FORTRAN can readily be adapted to accept variable names and statement labels beginning with previously illegal characters, such as $, or #, the conversion will be simplified. In some other cases, the implementation may permit the programmer to designate and reserve specific sets of labels and names for the conversion process. Neither of these, however, can be a limitation, and, in the general case, the source program must be completely pre-scanned so that blocks of legal, unused statement labels and variable names may be identified for use in the structure conversion.

As a part of the structure conversion phase, the original structure statements will be converted into comments, so that they can be retained for documentary purposes.

STRUCTURE DEFINITION

It is emphasized that, on a program by program basis, the programmer is free to define any number of structures (including none at all) that suits his (or her) convenience in the writing of the program. The programmer will be free to define his own structures, so long as he (or she) retains all of the essential parts for each structure. Thus, in what follows, the method of definition will be presented. Examples will be included for illustrative purposes only.

While no attempt is made to either prescribe or limit the forms of specific structures, certainly, all of the control structures occurring in the more common languages (e.g., ALGOL), expressed in any natural language using the Latin alphabet, should be definable for use in the dynamic FORTRAN programming system. In our dynamic system, words like "si", "alors", and "autre" could easily be used in place of "if", "then", and "else".

As indicated previously, the actual form of the structure definition depends upon the structure itself. The structures to be used are defined in a "structure division", placed in front of the main program. The first statement of each definition (beginning in Column 7) will be one of the following depending upon the structure:

```
STRUCTURE SEQUENTIAL
STRUCTURE PREDICATE
STRUCTURE REPETITIVE
STRUCTURE CASE
```

Each of these structures has its own, unique parts, which must be defined. Starting in Column 7, the component of the structure is indicated. That is followed by the statement to be used in the program to define the structure.

As an overall program organization, each program consists of a large sequential structure (which need not be explicitly defined). Within this structure, predicate, repetitive, and case structures, if needed, must be programmed in a form defined either in the structure section, or in standard FORTRAN. Structures may be nested within other structures. Indication of sequential structures is optional, except that they must be indicated explicitly when they are made up of more than one statement and are contained within predicate, case, or repetitive structures. At the other extreme, each statement is, by default, a sequential structure of one statement. There is no need to ever explicitly define a sequential structure of only one statement, even when it is inside of a predicate, repetitive, or case structure.

DEFINITION OF SEQUENTIAL STRUCTURES

The sequential structure is extremely trivial. To define a sequential structure or a sequential block, it is merely necessary to indicate its opening and its closing form. For example, a programmer might define the structure as follows:

```
STRUCTURE SEQUENTIAL
OPENING BEGIN
CLOSING END
```

This will cause the programming system to recognize groups of statements and/or structures between the defined OPENING and CLOSING statement brackets as a sequential structure. Each individual statement will be treated as a sequential structure without being so defined, and without having statement brackets. In the general case the defined OPENING statement will be converted into a comment, and otherwise ignored. The defined CLOSING statement will be converted into

a comment, and a CONTINUE with a legal label will be inserted just ahead of the converted END. The following special cases are recognized:

1. If the entire program is included as a sequential block, the defined CLOSING statement is converted into a conventional FORTRAN END statement. (It is anticipated, however, that non-executable statements, such as DIMENSION and FORMAT, will be placed outside of such blocks.)
2. If the structure is the THEN or ELSE part of a predicate structure, a PROCEDURE part of a replicative structure, or an ALTERNATIVE part of a case structure, a CONTINUE with a legal label is inserted just after the converted OPENING statement.
3. If the defined OPENING statement is labeled, a CONTINUE statement with this label is inserted just after the defined OPENING statement.
4. If the defined CLOSING statement is labeled, this label will be used on the CONTINUE inserted just before it.

DEFINITION OF PREDICATE STRUCTURES

The predicate structure consists of from two to four parts: a condition, an affirmative or condition—true alternative, an optional negative or condition—false alternative, and an optional closing. The condition itself functions as the opening of the structure. The two alternatives will be identified as sequential structures (possibly containing other structures). A closing may optionally be defined for the structure. If, however, no such closing is defined, the structure will be assumed to terminate at the end of the negative alternative, if any, or at the end of the positive alternative if there is no negative alternative.

As an example, consider:

STRUCTURE PREDICATE
OPENING IF (logical expression)
AFFIRMATIVE THEN (structure)
NEGATIVE ELSE (structure)
CLOSING END IF

(The use of parentheses is for generic purposes, i.e., any logical expression legal within FORTRAN, or any structure defined for the current program, may be used.)

The opening statement will be converted to a comment. A standard logical IF for the negative of the logical expression in the opening statement will be created to transfer to the negative alternative (or the closing if there is no negative alternative). This statement will carry the same label as the original opening in the structured form, if any. The positive alternative (a structure) will follow, in line, terminating with a converter generated GO TO the closing of the predicate structure. The negative alternative will then follow in line. A labeled CONTINUE will be inserted as the final statement (or closing) of the predicate structure. The user supplied indicators of the alternatives and the closing will be converted into comments.

It is noted that the requirement for a condition and an affirmative alternative structure precludes the interpretation of a standard FORTRAN logical IF as a defined structure. Use of the standard logical IF, followed by a single statement "then procedure" (other than the GO TO) is encouraged in the dynamic FORTRAN programming system. Such logical IF's possess all of the virtues of structure. Moreover, as part of standard FORTRAN, they require no further definition.

DEFINITION OF REPETITIVE STRUCTURES

The repetitive structure is, perhaps, the most difficult to define. This is a natural result of the fact that the repetitive structure permits a large number of variations. A controlled loop, whether or not it is arranged as a definite structure, has certain recognizable parts. These always include a body or procedure that is repeated many times, and a test that is performed many times, to determine whether or not to leave the loop. In some cases, the loop is controlled by a counter or index that must be initialized once and incremented many times. In some cases, there are data values to be initialized once. The testing may be done before each performance of the procedure, after each performance of the procedure or at a specific point within the procedure. In the latter case, there are, in effect, two procedures separated by a test, and organized in such a manner that there will be many performances of the two procedures, in order. (In what follows, we will not be limited as to the number of possible procedures to be repeated. We have implemented the full Omega-K structure of Bohm and Jacopini.[1]) We note, further, that there are two possible tests for leaving a loop: the loop may be continued UNTIL a certain condition becomes true, or it may be continued WHILE a certain condition remains true. (These are reverses of each other. In the former case, repeat on condition false, in the second repeat on condition true. Negating the condition permits switching of the test.)

Thus, in describing a repetitive structure, there is a mandatory opening section, an optional initialization, one or more procedures to be repeated, one or more tests for continuation or completion of the loop either before the first procedure, after the last procedure, or between any two procedures and finally a structure closing.

As an example consider:

STRUCTURE REPETITIVE
OPENING PERFORM
INITIAL ESTABLISH (structure)*

* The INITIAL part is optional.

CONTINUE or COMPLETE TEST (logical expression)‡

PROCEDURE (structure)**

CONTINUE or COMPLETE TEST (logical expression)‡

PROCEDURE (structure)**

CONTINUE or COMPLETE TEST (logical expression)‡

. .
. .
. .

CLOSING END PERFORM

During the structure conversion phase of the program, the opening statement will be converted to a comment, and the initialization part expanded as previously described. A labeled CONTINUE statement will be inserted either immediately after the initialization, if any, or after the opening. Each continue part or complete part will be preceded and followed by labeled CONTINUE statements, to be inserted if not provided by the programmer. (Two of these parts coming together, however, will not have a CONTINUE inserted between them.) The continue and complete parts, as logical expressions, will be incorporated into logical IF statements, as follows:

CONTINUE part
    IF (.NOT. expression) GO TO closing

COMPLETE part
    IF (expression) GO TO closing

The various procedure parts will be converted as previously described. Immediately before the closing component, a GO TO the beginning of the first procedure will be inserted by the converter. The closing itself will be converted into a labeled CONTINUE. As in the case of all structures, all statements requiring conversion will be converted into comments.

DEFINITION OF CASE STRUCTURES

The case structure involves the identification of an (integer-valued) arithmetic expression which functions as an index and a set of alternatives or choices to be executed depending upon the value of the index, i.e., if the index is 1, choice 1 only will be executed, etc. For purposes of implementation, the arithmetic expression functions as the opening of the structure. A closing part is mandatory.

As an example, consider:

STRUCTURE CASE

OPENING EXAMINE (arithmetic expression)

ALTERNATIVE n CASEn (structure)*

.
.
.

CLOSING END CASE

The opening statement itself is converted to a comment. If the arithmetic expression is not an integer variable name, an arithmetic assignment statement will be generated, setting the expression to an integer variable. The IFIX function will be inserted if needed. A computed GO TO statement will then be created, to send control to an appropriate alternative. The arithmetic assignment statement, if any, or the computed GO TO if no assignment statement is needed, will carry the same label as the original opening in the structured form, if any. Each of the alternatives will end with a GO TO the closing statement, which, itself, will be converted to a CONTINUE with a system generated label.

GENERAL PROVISIONS

During the process of interpreting the user supplied structure definition, the definition statements will be converted into comments. A blank comment statement will be inserted following every closing component.

All comments included in the original program will be retained during structure conversion.

It is generally recommended that structures be indented to facilitate their recognition. In using this dynamic programming system, no indentation rules are imposed upon the user. Indentation supplied by the user (if any) will be retained by the converter.

SAMPLES

In what follows, learning exercises will be shown written in a form for the FORTRAN programming system, and then converted into standard FORTRAN. They are intended purely as a demonstration of the concepts previously described.

Sample 1.—Table of roots of integer valued real numbers.

A. Source in Dynamic FORTRAN

STRUCTURE SEQUENTIAL
OPENING START
CLOSING FINISH
STRUCTURE REPETITIVE
OPENING REPEAT
INITIAL SET
PROCEDURE

** One PROCEDURE part required, the remainder are optional.
‡ One CONTINUE or COMPLETE part is required, the remainder are optional. A repetitive structure may be defined with both a continue and a complete part.

* Normally, there will be three or more of these components. The lower case n is used to indicate an integer number: 1,2, . . . maximum value.

Sample 1 continued

A. Source in Dynamic FORTRAN continued

```
COMPLETE TEST
CLOSING END REPEAT
STRUCTURE REPETITIVE
OPENING ITERATE
INITIAL SET
PROCEDURE
COMPLETE CONVERGE
CLOSING END ITERATE
DIMENSION Y(10)
START
WRITE (6,1001) (N,N=1,10)
REPEAT
  SET
  I=1
  START
  X=FLOAT(I)
  Y(1)=X
  REPEAT
    SET
    J=2
    START
    ITERATE
      SET
      START
      V=FLOAT(J)
      Z3=1.
      FINISH
      START
      Z=Z3
      Z1=(V-1.)*Z
      Z2=X/Z**(J-1)
      Z3=(Z1+Z2)/V
      FINISH
      CONVERGE
      ABS(Z-Z3) .LT. X*1.E-6
      END ITERATE
    Y(J)=Z3
    J=J+1
    FINISH
    TEST
    J .GT. 10
    END REPEAT
      WRITE (6,1002) Y
      I=I+1
      FINISH
      TEST
      I .GT. 50
      END REPEAT
    STOP
    FINISH
1001 FORMAT (1H1, 10(3X,5HROOT ,I2))
1002 FORMAT (1H ,10F10.6)
    END
```

B. Converted Standard FORTRAN

```
C     STRUCTURE SEQUENTIAL
C     OPENING START
C     CLOSING FINISH
C
C     STRUCTURE REPETITIVE
C     OPENING REPEAT
C     INITIAL SET
C     PROCEDURE
C     COMPLETE TEST
C     CLOSING END REPEAT
C
C     STRUCTURE REPETITIVE
C     OPENING ITERATE
C     INITIAL SET
C     PROCEDURE
C     COMPLETE CONVERGE
C     CLOSING END ITERATE
C
      DIMENSION Y(10)
C     START
      WRITE (6,1001) (N, N=1,10)
C     REPEAT
C       SET
        I=1
90001   CONTINUE
C       START
        X=FLOAT(I)
        Y(1)=X
C       REPEAT
C         SET
          J=2
90002     CONTINUE
C         START
C         ITERATE
C           SET
C           START
            V=FLOAT(J)
            Z3=1.
90003       CONTINUE
C           FINISH
90004       CONTINUE
C           START
            Z=Z3
            Z1=(V-1.)*Z
            Z2=X/Z**(J-1)
            Z3=(Z1+Z2)/V
90005       CONTINUE
C             FINISH
C           CONVERGE
            IF (ABS(Z-Z3) .LT. X*1.E-6) GO
              TO 90006
            GO TO 90004
90006       CONTINUE
C           END ITERATE
          Y(J)=Z3
          J=J+1
90007     CONTINUE
C         FINISH
```

Sample 1 continued

B. Converted Standard FORTRAN continued

```
C           TEST
            IF (J.GT. 10) GO TO 90008
            GO TO 90002
90008       CONTINUE
C           END REPEAT
          WRITE (6,1002) Y
          I=I+1
90009     CONTINUE
C         FINISH
C         TEST
          IF (I .GT. 50) GO TO 90010
          GO TO 90001
90010     CONTINUE
C         END REPEAT
        STOP
90011   CONTINUE
C       FINISH
 1001   FORMAT (1H1, 10(3X,5HROOT ,I2))
 1002   FORMAT (1H ,10F10.6)
        END
```

Sample 2.—Replacement sort of forty random numbers

A. Source in Dynamic FORTRAN

```
STRUCTURE SEQUENTIAL
OPENING BEGIN
CLOSING END
STRUCTURE PREDICATE
OPENING TEST
AFFIRMATIVE THEN
CLOSING END TEST
STRUCTURE REPETITIVE
OPENING LOOP
INITIAL SET
PROCEDURE
CONTINUE WHILE
CLOSING END LOOP
DIMENSION D(40)
BEGIN
WRITE (6,91)
LOOP
  SET
  I=1
  BEGIN
  CALL BGHT(D(I))
  WRITE (6,92) D(I)
  I=I+1
  END
  WHILE
  I .LE. 40
  END LOOP
WRITE (6,93)
LOOP
  SET
  I=1
  BEGIN
    LOOP
      SET
      BEGIN
      IMIN=I
      J=I+1
      END
      BEGIN
      TEST
        D(J) .LT. D(IMIN)
        THEN
        IMIN=J
        END TEST
      J=J+1
      END
      WHILE
      J .LE. 40
      END LOOP
    T=D(I)
    D(I)=D(IMIN)
    D(IMIN)=T
    I=I+1
    END
    WHILE
    I .LE. 39
    END LOOP
LOOP
  SET
  I=1
  BEGIN
  WRITE (6,92) D(I)
  I=I+1
  END
  WHILE
  I .LE. 40
  END LOOP
STOP
END
91 FORMAT (1H1, 38HSORT OF FORTY RANDOM
       NUMBERS, UNSORTED, //)
92 FORMAT (1H ,E15.8)
93 FORMAT (1H1, 36HSORT OF FORTY RANDOM
       NUMBERS, SORTED, //)
END
```

B. Converted Standard FORTRAN

```
C     STRUCTURE SEQUENTIAL
C     OPENING BEGIN
C     CLOSING END
C
C     STRUCTURE PREDICATE
```

Sample 2 continued

B. Converted Standard FORTRAN continued

```
C       OPENING TEST
C       AFFIRMATIVE THEN
C       CLOSING END TEST
C
C       STRUCTURE REPETITIVE
C       OPENING LOOP
C       PROCEDURE
C       CONTINUE WHILE
C       CLOSING END LOOP
C
        DIMENSION D(40)
C       BEGIN
          WRITE (6,91)
C         LOOP
C           SET
            I=1
90001       CONTINUE
C           BEGIN
              CALL BGHT(D(I))
              WRITE (6,92) D(I)
              I=I+1
90002         CONTINUE
C             END
C           WHILE
            IF (.NOT. (I .LE. 40)) GO TO 90003
            GO TO 90001
90003       CONTINUE
C           END LOOP
          WRITE (6,93)
C         LOOP
C           SET
            I=1
90004     CONTINUE
C         BEGIN
C           LOOP
C             SET
C             BEGIN
              IMIN=I
              J=I+1
90005         CONTINUE
C             END
90006       CONTINUE
C           BEGIN
C             TEST
              IF (.NOT. (D(J) .LT. D (IMIN) ))
                  GO TO 90007
C             THEN
              IMIN=J
90007         CONTINUE
C             END TEST
            J=J+1
90008       CONTINUE
C           END
C         WHILE
          IF (.NOT. (J .LE. 40)) GO TO 90009
          GO TO 90006
90009     CONTINUE
C         END LOOP
        T=D(I)
        D(I)=D(IMIN)
        D(IMIN)=T
        I=I+1
90010   CONTINUE
C       END
C       WHILE
        IF (.NOT. (I .LE. 39)) GO TO 90011
        GO TO 90004
90011   CONTINUE
        END LOOP
C       LOOP
C         SET
          I=1
90012     CONTINUE
C         BEGIN
            WRITE (6,92) D(I)
            I=I+1
90013       CONTINUE
C           END
C       WHILE
        IF (.NOT. (I .LE. 40)) GO TO 90014
        GO TO 90012
90014   CONTINUE
C       END LOOP
        STOP
90015   CONTINUE
C       END
91      FORMAT (1H1, "SORT OF FORTY RAN-
            DOM NUMBERS, UNSORTED" //)
92      FORMAT (1H ,F15.8)
93      FORMAT (1H1, "SORT OF FORTY RAN-
            DOM NUMBERS, SORTED" //)
        END
```

CONCLUSIONS

The foregoing discussion and samples demonstrate a method for moving the practice of programming into the nineteen seventies without abandoning FORTRAN. The following observations are made:

1. The user defined structures so dominate the program that, in its unconverted form, it is difficult to recognize it as FORTRAN at all.
2. The original source is free of GO TO statements, thus re-enforcing Dijkstra on the subject of that statement.[11]

REFERENCES

1. Bohm, C. and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with only Two Formation Rules," *Communications of the ACM*, Volume 9, No. 5, May 1966.

2. Dijkstra, E. W., "Notes on Structured Programming," in Dahl, Dijkstra, and Hoare, *Structured Programming*, Academic Press, 1972.
3. Naur, P., "Programming Languages, Natural Languages, and Mathematics," *Communications of the ACM*, Volume 18, No. 12, December 1975.
4. Ralston, A., Private Communication with this author, June 16, 1975.
5. X3.9.1966 *American Standard FORTRAN*, American Standards Association, Washington, 1966.
6. X3.10.1966 *American Standard Basic FORTRAN*, American Standards Association, Washington, 1966.
7. Wirth, N., "On the Design of Programming Languages," *Proceedings of the IFIP Congress 1974*, North Holland, 1974.
8. Johe, J. M., "Comments on the Topic 'Programming, and Its Implications on Programming Languages," *ACM '75 Proceedings of the Annual Conference.*
9. Archibald, J. A., Jr. and M. Katzper, "On the Preparation of Computer Science Professionals in Academic Institutions," *AFIPS Conference Proceedings*, Volume 43, 1974.
10. Dijkstra, E. W., "Go To Statement Considered Harmful," *Communications of the ACM*, Volume 11, No. 3, March 1968.
11. Ledgard, H. F. and M. Marcotty, "A Genealogy of Control Structures," *Communications of the ACM*, Volume 18, No. 11, November, 1975.
12. Fodor, J. A., T. G. Bever and M. F. Garrett, *The Psychology of Language*, McGraw-Hill Book Co., 1974.