FEATURE ARTICLES

Born-Again FORTRAN: FORTRAN 77

RICHARD E. BLOSS
Research Division, U.S. Industrial Chemicals Company, Cincinnati, Ohio 45237

Received March 6, 1984

Computer programming has become an important part of chemists' activities, both in the laboratory and in the office. A common topic of debate is which programming language is most suitable for chemical applications. Most chemists who program computers learned the FORTRAN IV language first and have continued to use it. Now that the number of available programming languages has grown, the perennial choice of the past, FORTRAN IV, has been the subject of some criticism. This paper suggests that the newer FORTRAN 77 should be recognized for its own merits as a powerful and flexible language and not passed over because of the shortcomings of earlier versions of FORTRAN.

INTRODUCTION

In the last decade it has become increasingly clear that today's scientists must be familiar, if not comfortable, with the use of computers in their routine work. The inclusion of microprocessors in most modern analytical instrumentation has brought with it the opportunity and obligation to put them to some productive use. When a scientist considers the purchase of new analytical equipment, he must not only be discriminating in choosing which unit will be best able to perform the chemical analysis reliably, but he must also be aware of the capability each unit possesses for digitizing, processing, and storing the raw data. Programmability, expandability, and software availability become key issues in the search for the right instrument.

The need to compute is not limited to the new equipment buyer. Mass and infrared spectral searches of computerized reference spectra are old and familiar applications. Nuclear magnetic resonance analysts must often rely on the use of the LAOCN-n[1] programs for analysis of complex spectra. NMR relaxation measurements can be related to molecular dynamics by using the program MOLDYN.[2] Even members of one of the last bastions of so-called pure chemistry, the synthetic organic chemists, are likely to be confronted with the need to compute. Cost-conscious managers may suggest numerical studies in place of performing scores of expensive exploratory syntheses. Such studies may include the use of pattern recognition or other numerical analysis of existing synthetic results[3] or programs that simulate or model the quantum mechanical behavior of molecules, such as CAMEO,[4] CNINDO,[5] MINDO/3,[6] or MNDO.[7] Of course, other examples of computational chemistry abound.

Clearly, there continues to be a great need for programming skills among the ranks of chemists. New chemical applications frequently are accompanied by a need for new computer programs. Even chemical applications for which software is available often demand some programming. For example, an analyst using the software that came with his Acme 5000 programmable gas chromatography system may simply not care for the way Acme decided to draw base lines. If Acme was generous enough to provide documentation and/or source code for their base-line routines and if the analyst understands the language in which this code was written, he may be able to modify the code to perform more to his liking. Alternatively, he may look at the existing code and, after deciding that it is "spaghetti-like" and intelligible only to the Acme programmer who wrote it (and maybe not even to him), begin to write a completely new base-line routine from scratch.

The preceding situation is only one of many which can induce a chemist who may have been perfectly happy doing chemistry to turn his attention to the study of a programming language. Many languages are available on mainframe computers, with a growing subset of them also now available on the popular microcomputers. Among the most favored for chemical applications today are APL, BASIC, FORTRAN, FORTH, and PASCAL. Languages such as ADA, C, and MODULA 2 also have received much attention but may not yet be as widely available on computers typically used by chemists.

CHOOSING THE LANGUAGE

Anyone who has had the experience of working with professional programmers has probably noticed that each programmer has a favorite language, the integrity, flexibility, and universal applicability of which he will defend staunchly. Chemists who have entered the programming arena often seem to be no different in this respect. Several opinions on the applicability of different programming languages for laboratory use have been published.[8-10] While there is no clear consensus on which language is preferred for all situations, there does seem to be a commonly held set of misconceptions about one language, FORTRAN. Formerly the unchallenged language of choice among the scientific community, FORTRAN is criticized more often now that other high-level languages are available. While it is true that many of the criticisms of FORTRAN were deserved at one time, many are no longer well founded. What is overlooked so often is that the FORTRAN that many chemists studied years ago has been supplanted by a newer, more powerful language. Its name is FORTRAN 77, and it meets the standards established in 1977 by the American National Standards Institute (ANSI). Critics of the old FORTRAN (which met the ANSI 1966 standards) have been slow to acknowledge the presence

0095-2338/85/1625-0001$01.50/0

Table I. Typical Reasons for Choosing a Particular Programming Language

1	prospective programmer already knows the language
2	prospective program user knows someone else who knows the language and who will help him or write the program for him
3	similar program exists that is written in the language
4	language is well suited (fast enough, flexible enough, etc.) to the type of problem being addressed
5	language is easy to learn and easy to use
6	programs written in the language are transferable, or nearly so, from one type of computer to another
7	language is highly regarded by the academic community and/or students of computer science

of the new. The remainder of this paper will discuss the differences between the two versions, with special attention being given to the needs of programmers in the scientific community.

Assuming for the moment that the choice of a language is not limited by commercial availability or purchase/lease price, why does a programmer choose one language over another? Table I lists seven common points of consideration in the selection of a language in what this author considers to be the most commonly demonstrated order of importance. Reasons one and two need no explanation. Many scientists and programmers are curious about other languages, but when the pressure of the moment dictates that a program be developed posthaste, the familiar language will almost always be chosen. The third reason is also very important. Professional programmers rarely write a new program from scratch, taking advantage, instead, of the benefits of modularity and reusing large sections of already working code. It is likely that these first three reasons contributed greatly to the proliferation of scientific programs written in FORTRAN, even after other powerful languages became available. "Lacking the compact power of APL, the intellectually satisfying elegance of ALGOL 68, the didactic incision of PASCAL, and the spurned universality of PL/I, FORTRAN survives, nay flourishes, thanks to a superior investmental inertia."[11]

Only after these first three points are addressed and discounted by the prospective programmer does he consider the last four. When control of laboratory events on a fast time frame is desired, a programmer may begin to pay close attention to comparisons of execution times of benchmark programs written in different languages. Otherwise, such studies are probably regarded out of curiosity. Of course, in some cases where speed and conservation of core memory are essential, no high-level language will suffice, and assembler language is the only choice. Assembler language programming is generally tedious and relatively challenging to learn, but the control over the computer provided by assembler language cannot be approached by most high-level languages. However, a new programmer will generally prefer a language that is easier to learn and that possesses convenient and powerful statements or structures. High-level languages, especially those with English-like commands, meet these needs. It may be argued that as the number of different control statements (DO WHILE, DO UNTIL, CASE, IF-THEN-ELSE, etc.) provided by a language increases, so does the desirability of the language. However, this enhancement is not without price. A language with a larger vocabulary requires more memory of both the computer and human types. Users of such notoriously large languages (e.g., PL/I) typically learn some subset of the language and make the commands they know serve any purpose that may arise. Nonetheless, languages that allow programmers to write structured, modular programs are very desirable.

FORTRAN IMPROVEMENTS

FORTRAN has been criticized for promoting unstructured programming and for its lack of convenience in the handling of nonnumeric data. References to FORTRAN's "strongly numeric philosophy" and "legendary disdain for text of any kind" have appeared.[12] It is likely that comments of this sort are directed only toward older versions of FORTRAN. However, it is important to note that these comments do not apply to FORTRAN 77.

A

```
      CHARACTER CAPS*26, LINE(100)*1
      INTEGER NUMCAP (26)
      DATA CAPS/'ABCDEFGHIJKLMNOPQRSTUVWXYZ'/, NUMCAP/26*0/
      READ(1,'(100A1)') LINE
      DO 1  I=1, 100
        DO  2 J=1, 26
            IF (LINE(I).EQ.CAPS(J:J) )  NUMCAP(J) = NUMCAP(J) + 1
2       CONTINUE
1     CONTINUE

      WRITE(6,'(3X, A1, 3X, I3)')  (CAPS(I:I), NUMCAP(I), I=1,26)
      END
```

B

```
VAR
  CHARACTER:CHAR;
  LETTER   :ARRAY(.'A'..'Z'.) OF INTEGER;
BEGIN
  REPEAT
  READ(CHARACTER);
  IF CHARACTER IN (.'A'..'Z'.) THEN
  LETTER(.CHARACTER.):=LETTER(.CHARACTER.)+1
UNTIL EOL(INPUT); END.
```

Figure 1. (A) FORTRAN 77 source code for counting the number of each upper-case letter in an input line; (B) PASCAL program segment for counting the number of each upper-case letter in an input line.

Character or text data can indeed be handled very directly with FORTRAN 77. An explicit character-type statement and an array of character operators and functions present programmers with an adequate set of tools for solving most problems involving text data. Notable among these tools are the functions for determining the location of substrings within character variables and for providing the integer equivalent or collating sequence number of a character argument. The ability to transfer data between character and numeric variables via internal read and write statements is also a valuable aid, especially in input error-checking applications.

In his comparison of FORTRAN to PASCAL, Marshall[12] cites the simplicity offered by the latter in the counting of the number of each capital letter read from a line of characters. A PASCAL solution to this problem (without any statements to generate output) is given,[13] and the reader is encouraged to imagine a comparable FORTRAN solution. Such a solution with FORTRAN 77 is presented in Figure 1A, with the cited PASCAL code shown for comparison in Figure 1B. The FORTRAN 77 solution lacks the elegance of the PASCAL version, especially with the latter's use of letters as array indexes. However, the FORTRAN 77 version is simple and provides an equally direct solution to this text-oriented problem.

One especially obnoxious characteristic of old versions of FORTRAN was ameliorated with the introduction of FORTRAN 77. FORTRAN critics often pointed to the language's reliance on GO TO statements for branching on logical and arithmetic conditions. The block IF-THEN-ELSE and ELSE-IF structures of FORTRAN 77 go a long way toward correcting this deficiency. With the exception of indicating the last statement of DO loops, FORTRAN 77 programmers

A

```
      REAL X
      F(X) = 1234. + 5.67*X + 8.90*X*X
      X = 0.
          DO 1 I = 1, 101
          Y = F(X)
          WRITE (6, 11)  X, Y
   11     FORMAT (F6.1,2X,F10.2)
    1     X = X + 0.50
         STOP
         END
```

B

```
      REAL X
      F(X) = 1234. + 5.67*X + 8.90*X*X
          DO 1 X = 0., 50., 0.5
    1   WRITE (6,'(F6.1,2X,F10.2)') X, F(X)
         END
```

Figure 2. (A) FORTRAN IV; (B) FORTRAN 77.

no longer need to use statement numbers if they prefer not to. These facts, taken in conjunction with FORTRAN's provision for true subroutines, allow considerable opportunity for programming using structured style.

The purpose of this paper is not to add yet another voice to the "my programming language is better than yours" argument. On the contrary, its intent is to erase an old image of FORTRAN that may be discouraging prospective programmers from considering a fine programming language that has many strengths and, yes, a few faults.

It is true that standard FORTRAN 77 does not have a DO WHILE construction, recursive or reentrant subroutines, or convenient access to bit manipulation. Furthermore, there is no facility for simultaneous control of concurrent events. However, the latter two defects can still be handled, when necessary, by using calls to subroutines written in an assembler language. Nevertheless, FORTRAN 77 is not just a new name for an old language. In fact, the improvements incorporated into FORTRAN 77, besides those already mentioned, are numerous and significant (see Table II).[14]

Advocates of structured programming usually insist on strict typing of variables, and this undoubtedly led to the rigidity of the academic model programming language PASCAL. Even those who favor PASCAL admit that its "type correspondence checks are carried to absurd limits."[12] FORTRAN 77, like its ancestor versions, provides an attractive alternative to such pedantry. FORTRAN 77 gives the programmer the choice of using explicit typing of all variables as demanded by structure dogma or of using the implicit or default types for integer and single-precision real variables. The availability of this freedom is very satisfying. How silly it would be to be forced to declare as integer every I or N used as a DO loop index variable!

While on the subject of FORTRAN 77 DO loops, another enhancement deserves to be mentioned. Prior to FORTRAN 77, if a FORTRAN programmer wanted to count from 0 to 50 by 0.5, perhaps to generate a table of function values over that range, he might have written code such as that shown in Figure 2A. In 1966-standard FORTRAN IV, DO loop indexes were not permitted to be real variables; hence, one would use an appropriate number of integers in the DO loop to allow the value of the real variable, incremented manually in a separate statement, to reach its desired maximum value. Comparable FORTRAN 77 code for this same task is shown in Figure 2B. Notice that in the FORTRAN 77 version, real constants (and expressions), including zero, are permissible in the DO statement parameters. Furthermore, the statements in a DO loop in FORTRAN 77 will not be executed at all if the initial index value equals or exceeds the final index control value. This "zero-pass" arrangement is a welcomed improvement over earlier versions of FORTRAN, in which all DO loop statements were executed at least once, despite the values held by the index control parameters.

Table II. Some of the Noteworthy Enhancements of FORTRAN 77 over FORTRAN IV[14]

1	completely blank lines are acceptable and are treated like comments
2	an explicit character type is permitted; functions for performing character–integer conversions and substring searches are provided
3	character assignment statements are allowed; this includes the ability to specify and assign the contents of one character variable to another
4	DIMENSION statements may include upper and lower bounds, including use of negative or zero subscripts
5	error conditions resulting from input/output operations may be monitored and acted upon from within the program
6	internal input/output statements permit transfer of data between numeric- and character-variable types
7	format expressions may be embedded in input/output statements
8	format expressions may include designated characters that control whether input blanks in numeric fields are interpreted as blanks or zeros
9	format expressions may include a designated character that ends format control if the input/output list is exhausted
10	block, IF, IF-THEN-ELSE, and ELSE-IF constructions are allowed
11	explicit OPEN, CLOSE, and INQUIRE statements are used for file handling
12	DO loop index variables may be real or integer
13	DO loop control parameters may be real or integer expressions and may be negative or zero
14	more than one named BLOCK DATA statement may be used
15	INCLUDE statements may be used to include blocks of code from a predefined system library at compile time

Another improvement in FORTRAN 77 demonstrated by Figure 2B is the ability to include input/output (I/O) format specifications in the I/O statements themselves. In this case, the WRITE and FORMAT statements used in the FORTRAN IV example were consolidated into a single WRITE statement in the FORTRAN 77 example. For those who prefer to separate FORMAT and I/O statements, the traditional I/O statement reference to the statement number of the FORMAT statement is still permissible.

The enhancements described above are truly noteworthy, as is one change that was not made. FORTRAN programmers still do not have to contend with the scourge of many other high-level programming languages—the statement-ending punctuation mark. Most programmers who favor punctuated languages such as PASCAL, PL/I, C, or SAS would agree (if caught in an unguarded moment of candor)

that omission of statement-ending punctuation is an all-too-frequent and especially insidious error.

CONCLUSIONS

Several differences between FORTRAN IV and FORTRAN 77 have been discussed in this paper in an attempt to call attention to the fact that the new version should not simply be regarded as the same as the old. Authors who use the generic language name "FORTRAN" are probably referring to FORTRAN IV or some other 1966-standard version. Such older languages served their purposes well but, by today's standards, are dreadfully limited. In contrast, FORTRAN 77 has the benefit of many powerful enhancements, including character-handling and improved logical structures. FORTRAN 77 also provides upward object module compatibility for programs compiled with standard FORTRAN IV compilers.

It has not been the intent of this paper to convince readers that any version of FORTRAN is the language of choice for all applications. For many applications it may be, but this is very much a personal decision, as mentioned earlier. It is undeniable that an abundance of packaged software and functional subroutines written in FORTRAN IV already exists. This alone will undoubtedly assure FORTRAN 77 a continued existence for many years because of FORTRAN's upward compatibility. The point is, though, that FORTRAN 77 is a truly powerful and flexible language that can stand on its own merits. While everyone may not agree with Post's declaration "Real programmers use FORTRAN,"[15] we definitely should not form our opinion of the language on the basis of outdated information.

ACKNOWLEDGMENTS

I thank Michael Benson and Mark Faulhaber for many stimulating discussions on this and other topics, Mae Cantor and Rusty McCracken for typing the manuscript, and Michelle Rudy for performing literature searches.

REFERENCES AND NOTES

(1) See, for example, "Quantum Chemical Program Exchange Catalog of Available Programs"; Quantum Chemistry Program Exchange, Department of Chemistry, Indiana University: Bloomington, IN, 1982; Section IV, Vol. XIV, pp 2, 16.
(2) Craik, D. J.; Kumar, A.; Levy, G. C. "MOLDYN: A Generalized Program for the Evaluation of Molecular Dynamics Models Using Nuclear Magnetic Resonance Spin-Relaxation Data". *J. Chem. Inf. Comput. Sci.* **1983**, *23*, 30.
(3) Wold, S.; Dunn, W. J., III "Multivariate Quantitative Structure-Activity Relationships (QSAR): Conditions for Their Applicability". *J. Chem. Inf. Comput. Sci.* **1983**, *23*, 6, and references cited therein.
(4) Salatin, T. D.; Jorgenson, W. L. "Computer-Assisted Mechanistic Evaluation of Organic Reactions. 1. Overview". *J. Org. Chem.* **1980**, *45*, 2043.
(5) Pople, J. A.; McIver, J. W.; Ostlund, N. S. "Self-Consistent Perturbation Theory. II. Nuclear-Spin Coupling Constants". *J. Chem. Phys.* **1968**, *49*, 2965.
(6) Bingham, R. C.; Dewar, M. J. S.; Lo, D. H. "Ground States of Molecules. XXV. MINDO/3. An Improved Version of the MINDO Semiempirical SCF-MO Method". *J. Am. Chem. Soc.* **1975**, *97*, 1285.
(7) Dewar, M. J. S.; Thiel, W. "Ground States of Molecules. 38. The MNDO Method. Approximations and Parameters". *J. Am. Chem. Soc.* **1977**, *99*, 4899.
(8) Avery, J. P. "Using Microcomputers in the Laboratory". *J. Chem. Inf. Comput. Sci.* **1983**, *23*, 1.
(9) Dessy, R. E., Ed. "Languages for the Laboratory: Part I". *Anal. Chem.* **1983**, *55*, 650A.
(10) Dessy, R. E., Ed. "Languages for the Laboratory: Part II". *Anal. Chem.* **1983**, *55*, 756A.
(11) Kelly-Bootle, S. "The Devil's D P Dictionary"; McGraw-Hill: New York, 1981; p 54.
(12) Marshall, J. C. "Microcomputer Software. 1. After FORTRAN". *J. Chem. Inf. Comput. Sci.* **1983**, *23*, 91.
(13) From example given by Anderson, R. W. "From Basic to Pascal"; TAB Books: Blue Ridge Summit, PA, 1982; p 9.
(14) DPD Education Staff Services, Publishing/Media Support, Education Center, IBM Corp. "VS FORTRAN Enhancement Features"; International Business Machines: 1981; Appendix A, Publication SR20-4722-0.
(15) Post, E. "Real Programmers Don't Use PASCAL". *Datamation* **1983**, *29* (7), 263.