**Invited Paper:**

# The Seven Ages of Fortran

**Michael Metcalf**
**Manfred-von-Richthofenstrasse 15**
**12101 Berlin, Germany**
*michaelmetcalf@compuserve.com*

## Abstract

When IBM's John Backus first developed the Fortran programming language, back in 1957, he certainly never dreamt that it would become a world-wide success and still be going strong many years later. Given the oft-repeated predictions of its imminent demise, starting around 1968, it is a surprise, even to some of its most devoted users, that this much-maligned language is not only still with us, but is being further developed for the demanding applications of the future. What has made this programming language succeed where most slip into oblivion?

One reason is certainly that the language has been regularly standardized. In this paper we will trace the evolution of the language from its first version and though six cycles of formal revision, and speculate on how this might continue.

Now, modern Fortran is a procedural, imperative, compiled language with a syntax well suited to a direct representation of mathematical formulas. Individual procedures may be compiled separately or grouped into modules, either way allowing the convenient construction of very large programs and procedure libraries. Procedures communicate via global data areas or by argument association. The language now contains features for array processing, abstract data types, dynamic data structures, object-oriented programming and parallel processing.

**Keywords:** array processing, data abstraction, object-oriented programming, optimization, history of computing.

## Language evolution

### 1. The First Age: Origins

In the early days of computing, programming was tedious in the extreme – every tiny step had to be coded as a separate machine instruction, and the programmer had to be familiar with the intimate details of the computer's operation. Spurred by a perceived economic need to provide a form of 'automatic programming' to allow efficient use of manpower and computers, Backus proposed, at the end of 1953, to begin the development of the Fortran programming language (the name being a contraction of FORmula TRANslation). The overriding objective of the development team was to produce a compiler that would produce efficient object code comparable to that of hand-written assembly code.

Fortran came as a breakthrough. Instead of writing some obscure hieroglyphics, say as an instruction to divide two variables A and B and to save the result in C, the programmer could now write a more intelligible and natural statement, namely

```
C = A/B
```

This is called *expression abstraction*, because mathematical expressions could be written more-or-less as they appear in a textbook. Herein lay the secrets of Fortran's initial rapid spread: scientists could write programs to solve problems themselves, in a familiar way and with only limited recourse to professional programmers, and that same program, once written, could be transported to any other computer which had a Fortran compiler. The first version, now known as FORTRAN I, contained early forms of constructs that have survived to the present day: simple and subscripted variables, the assignment statement, a do-loop, mixed-mode arithmetic, and input/output (I/O) specifications.

Many novel compiling techniques had to be developed, and it was not until 1957 that the first compiler was released to users of the target machine, the IBM 704. First experience showed that, indeed, it increased programmer efficiency and allowed scientists and engineers to program easily for themselves. The source form and syntax liberated programmers from the rigid input formats of assembly languages. Fortran was an immediate success.

Ease of learning and stress on optimization are two hallmarks of Fortran that have contributed to its continued popularity.

Based on the experience with FORTRAN I, it was decided to introduce a new version, FORTRAN II, in 1958. The crucial differences between the two were the introduction of subprograms, with their associated concepts of shared data areas, and separate compilation. FORTRAN II became the basis for the development of compilers by other manufacturers. A more advanced version was developed for the IBM 704 – FORTRAN III – but it was never released.

## 2. The Second Age: FORTRAN 66

In 1961, an IBM users' organization requested from IBM a new version, now called FORTRAN IV, which contained type statements, the logical-`if` statement, the possibility to pass procedure names as arguments, and the `data` statement and `block data` subprogram. Some original features, such as device-dependent I/O statements, were dropped.

FORTRAN IV was released in 1962 and quickly became available on other machines, but often in the form of a dialect. Indeed, the proliferation of these dialects led an American Standards Association (ASA) Working Group to develop a standard definition for the language. In 1966, a standard for FORTRAN was published, based on FORTRAN IV. This was the first programming language to achieve recognition as a national, and subsequently international (ISO, Geneva), standard, and is now known as FORTRAN 66.

FORTRAN 66 was made available on almost every computer made at that time, and was often pressed into service for tasks for which it had never been designed. Thus began a period during which it was very popular with scientists, but newer, more modern languages were appearing, including Algol 60, whose 'superior' concepts led to predictions that it would rapidly replace 'old-fashioned` Fortran because of the latter's limitations. It became increasingly criticized, especially by academic computer scientists.

## 3. The Third Age: FORTRAN 77

The permissiveness of the FORTRAN 66 standard, whereby any extensions could be implemented on a given processor so long as it still correctly processed a standard-conforming program, led again to a proliferation of dialects. These dialects typically provided much-needed additional features, such as bit handling, or gave access to hardware-specific features, such as byte-oriented data types. Since the rules of ASA's successor, the American National Standards Institute (ANSI), required that a standard be reaffirmed, withdrawn or revised after a five-year period has elapsed, the reformed FORTRAN committee, X3J3, decided on a revision. This was published by ANSI, and shortly afterwards by ISO, in 1978, and became known as FORTRAN 77. The new standard brought with it many new features, for instance the `if…then…else` construct (from the push for 'structured programming'), a character data type, and much enhanced I/O.

The new language was rather slow to spread. This was due in part to certain conversion problems and also to the decision of one large manufacturer, IBM, not to introduce a new compiler until 1982. It was thus only in the mid-1980s that FORTRAN 77 finally took over from FORTRAN 66 as the most used version. Ultimately, it became a hugely successful language for which compilers were available on every type of computer from the PC to the mighty Cray. Programs written in FORTRAN 77 were routinely used to perform such diverse calculations as designing the shapes of airplane fuselages, predicting the structures of organic molecules, and simulating the flow of winds over mountains.

Algol is now a dead language, however it begat descendents, most notably Pascal and Ada, and these too, in their time, together with IBM's PL/1, were variously considered to be about to deliver the *coup-de-grâce* to FORTRAN. But the new standard lent it a new vigour that allowed it to maintain its position as the most widely used scientific applications language of the time. However, it began to yield its position as a teaching language.

The entire issue of the journal [1] is devoted to papers on the early history of Fortran.

## 4. The Fourth Age: The battle for Fortran 90

As computers doubled in power every few years, and became able to perform calculations on many numbers simultaneously, by the use of processors running in parallel, and as the problems to be solved became ever more complex, the question arose as to whether FORTRAN 77 was still adequate. (And there were a large number of user requests left over that it had not been possible to include in it.) Programs of over a million lines became commonplace, and managing their complexity and having the means to write them reliably and understandably – so that they produce correct results and could later be modified – were desperately required.

Thus began the battle over Fortran 90.[1] Fortran had been attacked by computer scientists on two grounds. One was because of its positively dangerous aspects, for instance the lack of any inherent protection against overwriting the contents of memory in the computer, including the program instructions themselves! The other was its lack of indispensible language features, such as the ability to control the logical flow through a program in a clearly structured manner. On the other hand, Fortran had always been a relatively easy language to learn and that, combined with its emphasis on efficient, high-speed processing, had kept it attractive to many busy scientists. Thus, the standards committees were faced with the almost impossible task of modernising the language and making it safer to use, whilst at the same time

[1] And a change to lower-case spelling.

keeping it 'Fortran-like' and efficient. Fortran 90 was the answer.

There were other justifications for continuing to revise the definition of the language. As well as standardizing vendor extensions, there was a need to respond to the developments in language design that had been exploited in other languages, such as APL, Algol 68, Pascal, Ada, C and C++. Here, X3J3 could draw on the obvious benefits of concepts like data hiding. In the same vein was the need to begin to provide an alternative to dangerous storage association, to abolish the rigidity of the outmoded source form, and to improve further on the regularity of the language, as well as to increase the safety of programming in the language and to tighten the conformance requirements. To preserve the vast investment in Fortran 77 codes, the whole of Fortran 77 was retained as a subset. However, unlike the previous standard, which resulted almost entirely from an effort to standardize *existing practices*, the Fortran 90 standard was much more a *development* of the language, introducing features that were new to Fortran, although based on experience in other languages. This tactic, in fact, proved to be highly controversial, both within the committee and with the wider community. Vested interests got in on the act, determined, depending on their persuasion, and in particular on whether they were users or vendors, either to extend Fortran to cope better with new computers and new problem domains or to stop the whole process in its tracks. The technical and political infighting reached legendary proportions. It was not until 1991, after much vigorous debate and thirteen years' work, that Fortran 90 was finally published by ISO.

It introduced a new notation that allows arrays of numbers, for instance matrices, to be handled in a natural and clear way, and added many new built-in facilities for manipulating such arrays, for example, to add together all the numbers in an array, a single command (`sum`) is all that is required. The use of the array-handling facilities made scientific programming simpler, less error prone and, on the most powerful computers whose hardware can handle vectors of numbers, potentially more efficient than ever.

To make programs more reliable, the language introduced a wealth of features designed to catch programming errors during the early phase of compilation, when they can be quickly and cheaply corrected. These features included new ways of structuring programs and the ability to ensure that the components of a program, the subprograms, 'fit together' properly. For instance, Fortran 90 makes it simple to ensure that an argument mismatch can never arise as it enables programmers to construct verifiable interfaces between subprograms.

In summary, the main features of Fortran 90 were, first and foremost, the array language and data abstraction. The former is built on whole array operations and assignments, array sections, intrinsic procedures for arrays, and dynamic storage. It was designed with optimization in mind. The latter is built on modules and module procedures, derived data types, operator overloading and generic interfaces, together with pointers. Also important were the new facilities for numerical computation, including a set of numeric inquiry functions, the parameterization of the intrinsic types, new control constructs – `select case` and new forms of `do`, internal and recursive procedures and optional and keyword arguments, improved I/O facilities, and many new intrinsic procedures. Last but not least were the new free source form, an improved style of attribute-oriented specifications, the `implicit none` statement, and a mechanism for identifying redundant features for subsequent removal from the language. The requirement on compilers to be able to identify syntax extensions, and to report why a program had been rejected, was also significant. The resulting language was not only a far more powerful tool than its predecessor, but a safer and more reliable one too. Storage association, with its attendant dangers, was not abolished, but rendered unnecessary. Indeed, experience showed that compilers detected errors far more frequently than before, resulting in a faster development cycle. The array syntax and recursion also allowed quite compact code to be written, a further aid to safe programming. Fortran 90 also allowed programmers to tailor data types to their exact needs. Another advance was the language's new ability to structure program data into arbitrarily complex patterns – lists, graphs, trees, etc. – and to manipulate these structures conveniently. This is achieved through the use of pointers. A related feature was the ability to allocate storage for program data dynamically.

After this revision, Fortran became, it must be admitted, a different language, as the entire issue of the journal [2], which is devoted to various aspects of the development of Fortran 90, shows.

## 5. The Fifth Age: A minor revision, Fortran 95

Following the publication of Fortran 90, two further significant developments concerning the language occurred. The first was the continued operation of the two standards committees, J3 (as X3J3 became known) and the international WG5, and the second was the founding of the High Performance Fortran Forum (HPFF).

Early on in their deliberations, the committees decided on a strategy whereby a minor revision of Fortran 90 would be prepared by the mid-1990s and a further revision by about the year 2000. The first revision, Fortran 95, is the subject of this section.

The HPFF was set up in an effort to define a set of extensions to Fortran, such that it would be possible to write portable, single-threaded code when using parallel computers for handling problems involving large sets of data that can be represented by regular grids. This version of Fortran was to be known as High Performance Fortran (HPF), and Fortran 90 was chosen as the base language. Thus, HPF was a superset of Fortran 90, the main extensions being expressed in the form of directives. However, it did become necessary also to add some additional syntax, as not all of the desired features could be accommodated in the form of directives.

It was evident that, in order to avoid the development of divergent dialects of Fortran, it would be desirable to include the new syntax defined by HPF in Fortran 95 and, indeed, these features were the most significant new ones that Fortran 95 introduced. The other changes consisted mainly of what are known as corrections, clarifications and interpretations. Only a small number of other pressing but minor language changes were made.

A new ISO standard, replacing Fortran 90, was adopted in 1997.

## 6. The Sixth Age: Fortran 2003

Without a break, standardization continued, and the following language standard, Fortran 2003, was published, somewhat delayed, in 2004. The major enhancements were:

- Derived type enhancements: parameterized derived types, improved control of accessibility, improved structure constructors, and finalizers.
- Object-oriented programming support: type extension, inheritance, polymorphism, dynamic type allocation, and type-bound procedures.
- Data manipulation enhancements: deferred type parameters, the `volatile` attribute, explicit type specification in array constructors and allocate statements, pointer enhancements, extended initialization expressions, and enhanced intrinsic procedures.
- Input/output enhancements: asynchronous transfer, stream access, user-specified transfer operations for derived types, user-specified control of rounding during format conversions, named constants for pre-connected units, the `flush` statement, regularization of keywords, and access to error messages.
- Procedure pointers.
- Support for the exceptions of the IEEE Floating-Point Standard (IEEE 1989).
- Interoperability with the C programming language.
- Support for international usage: access to ISO 10646 4-byte characters and the choice of decimal or comma in numeric formatted input/output.
- Enhanced integration with the host operating system: access to command-line arguments, environment variables, and processor error messages.

In addition, there were numerous minor changes but Fortran 2003 was essentially upwards compatible with the Fortran 95 standard that it replaced. The enhancements had, after all, been developed in response to demands from users and to keep Fortran relevant to the needs of programmers, without losing the vast investment in existing programs.

### Related standards

No Fortran standard up to and including Fortran 2003 included any significant feature intended directly to facilitate parallel programming. Rather, this has had to be achieved through the intermediary of *ad hoc* industry standards, in particular HPF, MPI, OpenMP and Posix Threads.

HPF directives take the form of Fortran comment lines that are recognized as such only by an HPF processor. An example is

```
!HPF$ ALIGN WITH b :: a1, a2, a3
```

to align three conformable (matching in shape) arrays with a fourth, thus ensuring locality of reference. Further directives allow, for instance, aligned arrays to be distributed over a set of processors.

MPI is a library specification for message passing. OpenMP supports multi-platform, shared-memory parallel programming and consists of a set of compiler directives, library routines, and environment variables that determine run-time behaviour. Posix Threads is again a library specification, for multithreading.

MPI and OpenMP have both become widespread, but HPF has ultimately met with little success.

## 7. The Seventh Age: Fortran 2008

Notwithstanding the fact that Fortran 2003-conformant compilers have been very slow to appear, the standardization committees proceeded with yet another standard, Fortran 2008. Its single most important new feature is coarray handling (described below). Further, the `do concurrent` form of loop control and the `contiguous` attribute are introduced. Other major new features include: sub-modules, enhanced access to data objects, enhancements to I/O and to execution control, and more intrinsic procedures, in particular for bit

processing. Fortran 2008 was published in 2010 [3], and is the current standard.

## Fortran concepts

Programming languages have many features in common. In this section some that represent Fortran's special strengths are briefly outlined. Its object-oriented features are, however, omitted, but these and all the other features are fully described in [4].

### 1. Array processing

Fortran 95 allows variables and function results to be array valued. Such objects can be manipulated in much the same way as scalar objects. Thus, given the declaration[2]

```
real, dimension(10) :: a, b, c, x
```

specifying four real, conformable arrays, `a`, `b`, `c` and `x`, we might write:

```
x = (-b + sqrt(b**2 - 4.0*a*c)/(2.0*a)
```

to solve a set of quadratic equations rather than just one. Scalar values within array expressions are 'broadcast' across the whole extent of the array variables. In this example, the use of the `where` (masked assignment) construct would be necessary in order to avoid division by zero, if this is likely to occur:

```
where (a /= 0.0)
   x=(-b + sqrt(b**2 - 4.0*a*c)/(2.0*a)
elsewhere
   x = -huge(0.0) !large negative real
end where
```

An array assignment expressed with the help of indices is provided by the `forall` statement and construct. An example is

```
   forall(i = 1:n) x(i, i) = s(i)
```

where the individual assignments may be carried out in any order, and even simultaneously. In

```
   forall(i=1:n, j=1:n, y(i,j)/=0.)  &
                  x(j,i) = 1.0/y(i,j)
```

the assignment is subject also to a masking condition., once again to avoid division by zero. Any procedure referenced within a `forall` statement or construct must have the `pure` attribute to ensure that it has no side effects that could cause the result to depend on the order of execution.

The `sqrt` function seen above is used as an *elemental* function: although defined in terms of scalars it returns an array-valued result for an array-valued argument. Many intrinsic procedures are elemental, and a user-written procedure that is pure may be made elemental by adding the `elemental` keyword to its header line and by following certain rules.

An array need not necessarily be specified with a fixed size. If `a` is an array dummy argument, it may be declared as an assumed-shape array

```
real, dimension(:, :) :: a
```

where the actual array bounds are transmitted between the two procedures at run time.

Further, an array that is local to a procedure may be specified as an automatic array whose bounds depend on another argument, as in

```
real, dimension(size(a)) :: work
```

to define an array `work` whose size depends on that of another array `a`.

Lastly, storage may be allocated dynamically to an array at run time. Given a specification as in:

```
real, dimension(:, :), allocatable :: g
```

we may write

```
allocate(g(50, 100))
```

to give the required space to the array at run-time. The space may later be deallocated and then allocated afresh. The `allocate` and `deallocate` statements are equally useful for arrays that have the `pointer` attribute, in particular for dynamic arrays that are components of a derived data type.

Given a rank-two array that has, one way or another, been given appropriate bounds, we may reference a single (scalar) element using a subscript notation as in

```
grid(9, 15)
```

A subsection of the array may be referenced using a triplet notation as in

```
grid(1:10, 10:100:10)
```

which is an array-valued subobject that may, in turn, appear in array expressions and assignments. It is

[2] Within the code extracts, Fortran keywords will be written in bold face, in order to distinguish them from variable names.

that subsection of `grid` that consists of its first ten elements in the first dimension and every tenth element in the second. It is a ten-by-ten, rank-two array.

An array-valued constant is known as an array constructor. It has a variety of forms, a simple one being shown in

```
grid(1:5,10) = (/1.0,2.0,3.0,4.0,5.0/)
```

A pointer may be used as a dynamic alias to an array or to an array subobject. If we add the `target` attribute to the specification of `grid`, and define an appropriate pointer array as

```
real, dimension(:), pointer :: window
```

then the pointer assignment

```
window => grid(0:9, 1)
```

makes `window` a rank-one array of length ten.
The many array functions defined by the standards are an important and integral part of the array-processing language.

## 2. Coarrays

The objective of coarrays is to allow the simultaneous processing of arrays on multiple processors. In this model, not only is data distributed over processors, as in an SIMD (Single Instruction Multiple Data) model, but also work, using the SPMD (Single Program Multiple Data) model. The syntax required makes only a small impact on the appearance of a program.

Data distribution is achieved by specifying the relationship among memory *images*. Any object declared *without* using the corresponding syntax exists independently in all the images and can be accessed only from within its own image. Objects specified *with* this syntax have the additional property that they can be accessed directly from any other image. Thus, the statement

```
real, dimension(512) [*] :: a, b
```

specifies two coarrays, `a` and `b`, that have the same size (512) in each image. Execution by an image of the statement

```
a(:) = b(:)[j]
```

causes the array `b` from image `j` to be copied into its own array `a` (where square brackets are the notation used to access an object on another image). On a shared-memory machine, an implementation of a coarray might be as an array of a higher dimension. On a distributed memory machine with one physical processor per image, a coarray will probably be stored at the same address in each physical processor.

Work is distributed as images, which are copies of the program each of which has a separate set of data objects and a separate flow of control. The number of images is a fixed value that is available at execution time via an inquiry function, `num_images`. The images execute asynchronously and the execution path in each may differ. The programmer has access to the image index via the `this_image` function. When synchronization between two images is required, use can be made of a set of intrinsic synchronization procedures, such as `sync_lock` or `lock`. Using these, it is possible to avoid race conditions whereby one image alters a value still required by another, or one image requires an altered value that is not yet available from another. Between synchronization points an image has no access to the fresh state of any other image. Any flushing of temporary memory, caches or registers is handled implicitly by the synchronization mechanisms themselves. Thus, a compiler can safely take advantage of all code optimizations on all processors between synchronization points without compromising data integrity. Where it might be necessary to limit execution of a code section to just one image at a time, a critical section may be defined using a `critical… end critical` construct.

The *codimensions* of a coarray are specified in a similar way to the specifications of assumed-size arrays, and coarray sub-objects may be referenced in a similar way to sub-objects of normal arrays.

The following example shows how coarrays might be used to read values in one image and distribute them to all the others:

```
real :: value[*]
…
if(this_image() == 1) then
! Only image 1 executes this construct.
    read(*, *) value
    do image = 2, num_images()
       value[image] = value
    end do
end if
call sync_all()
! Execution on all images pauses at
! this point until all images have
! reached it.
```

Coarrays can be used in most of the ways that normal arrays can, the most notable restrictions being that they cannot be automatic arrays, cannot be used for a function result, cannot have the

pointer attribute, and cannot appear in a pure or elemental procedure.

### 3. Abstract data types and data structures

When an abstract data type has been defined, for instance by

```
  type interval
    real :: lower, upper
  end type interval
```

it is further possible to define the meanings associated with operations and assignments on objects of that type, or between an object of that type and an object of another derived or intrinsic type. The usual mechanism is to specify functions (for operations) or subroutines (for assignments) that perform the necessary tasks, and to place these in a module that can be accessed to gain access to the types, the operations and the assignments. An example that defines a type suitable for interval arithmetic, defines the operation to perform addition between two scalar objects of that type, and defines assignment of a real object to an object of type interval is:

```
module intervals
  type interval
    real :: lower, upper
  end type interval
  interface operator(+)
    module procedure add_interval
  end interface
  interface assignment(=)
    module procedure interval_from_real
  end interface
contains
  function add_interval(a,b)
    type(interval) :: add_interval
    type(interval), intent(in) :: a, b
    add_interval%lower=a%lower+b%lower
    add_interval%upper=a%upper+b%upper
  end function add_interval
  subroutine interval_from_real(a,b)
    type(interval), intent(out) :: a
    real, intent(in)            :: b
    a%lower = b
    a%upper = b
  end subroutine interval_from_real
end module intervals
```

A snippet of code that makes use of the facilities thus defined would be:

```
program demo
   use intervals
   real           :: a = 1.0
   type(interval) :: b, c
   b = a     ! defined assignment
   c = a     ! defined assignment
   c = b + c ! defined operation
   print *, a, b, c
end program demo
```

This main program and the module intervals together form a complete, executable program.

Derived-data types may contain components that have the pointer attribute. This allows the construction of data structures of arbitrary complexity. If the elements of a sparse vector are to be held as a chain of variables, a suitable data type would be

```
type entry
   real                  :: value
   integer               :: index
   type(entry),pointer   :: next=>null()
end type entry
```

A chain can then be specified by

```
type(entry), pointer :: chain
```

and the first variable can be defined by, for example,

```
allocate(chain)
chain%value = 1.0
chain%index = 10
```

Normally, such a list would be manipulated with the aid of additional pointers that reference, for instance, its first and current entries, and with utility procedures for adding and removing entries, *etc*. Once again, it would be usual to package the type and the procedures that manipulate the list into a module.

## The status of Fortran

### 1. Challenges from other languages

Fortran has always had a slightly old-fashioned image. In the 1960s, the block-structured language Algol was regarded as superior to Fortran. In the 1970s the more powerful PL/1 was expected to replace Fortran. Algol's successors Pascal and Ada caused Fortran proponents some concern in the 1980s. Meanwhile, it continued successfully as the workhorse of scientific computing. However, by the late 1980s, two developments did begin seriously to impinge on Fortran's predominance in this field: Unix and object orientation.

Unix brought with it the highly-successful general-purpose language C, which was further developed into C++, an object-oriented language. C is widely used for all levels of system programming and made inroads into Fortran's traditional numerical computing community. C++ came to dominate many programming applications especially those requiring sophisticated program interfaces. Another object-oriented language, Java, has also come into widespread use.

Fortran's particular advantages as a high-end numerical language, especially where arrays are the main data object and/or where complex arithmetic is involved, remain. It is able to attain the highest achievable optimization, mainly because multi-dimensional arrays are 'natural' objects and because its pointers are highly constrained. Nevertheless, whether modern Fortran will, in the long term, be able to withstand the immense pressure from other languages remains an open question. However, there is every sign that Fortran continues to be used to tackle major scientific computing problems, and will long remain a living memorial to the early pioneers. Indeed, at a Workshop on Software in High-Energy Physics in 1982, I predicted that: *"Fortran is likely to remain into the next century as, at the very least, a special-purpose scientific and numerical language for large-scale, computing-intensive applications and, strengthened especially by its array capabilities, will be one of a small range of widely-used languages in general use"*. This turned out to be not too far from the truth!

### 2. The international Fortran community

Fortran is an international language both in the sense that it used throughout the world, and also in that the community of international users has, over the last 30 years, actively participated in the development of the standards. Furthermore, the Internet and the World-Wide Web have facilitated the development of international user communities, for instance the newsgroup *comp.lang.fortran*, and the discussion group at *www.jiscmail.ac.uk/cgi-bin/webadmin?A0=comp-fortran-90*

These groups are important in the dissemination of Fortran news, such as announcements of new compilers, and as sources of help and advice to users in general. The ACM publishes *Fortran Forum,* a special interest publication on Fortran with an international readership and containing articles on Fortran language developments and user experience (see *www.sigplan.org*). A table detailing the progress by various vendors in their implementations of the latest two standards is maintained at *www.fortranplus.co.uk/resources/fortran_2003_2008_compiler_support.pdf.*

We thus see that there is a healthy user community, even if the language now occupies, in contrast to the past, only a niche in the world of programming, but one nevertheless concerned with large and important applications. Long may it continue!

## References

[1] Annals of the History of Computing. Vol. 6, No. 1 (1984).

[2] Computer Standards & Interfaces, Vol. 18 (1996).

[3] ISO/IEC 1539-1 : 2010. ISO, Geneva, Switzerland.

[4] Metcalf, M., Reid. J. and Cohen, M. (2011). *Modern Fortran Explained*. Oxford University Press, Oxford and New York.