EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

The F Programming Language

Ralph Frisbie
Ventura College, Ventura, CA
Richard Hendrickson
Imagine1, Inc., Albuquerque, NM
Michael Metcalf
CERN, Geneva, Switzerland

Abstract

We introduce a new programming language, F, that is intended for the teaching of modern concepts. Strong typing and a strict syntax facilitate teaching and learning. The language is available in optimized form for all major platforms and some teaching experience has already been gained.

1 Why a new language?

Is the teaching of programming languages in a crisis? At least to judge from the interviews in a recent issue of ACM SIGPLAN Notices [1], there is some divergence of views among experts on what is desirable and where the field is headed. However, among the points made in the article, specifically on programming languages, are the need for strong typing, the difficulty of jumping straight into object orientation, the desirability of abstract data types, the need to avoid teaching too many concepts that are too little understood, the fact that C++ is too big and complex and that PASCAL appears to be of no long-term use, the need to concentrate on fundamentals, and a noticeable affection for Scheme.

Apart from this final point, we would like to think that the recent development of the F programming language satisfies all of these criteria. It is strongly typed and has abstract data types. It is defined in terms of a BNF that is strict and leads to early detection of most errors. All interprocedural interfaces are explicit. A modular programming style is encouraged, and the existence of pointers provides support for data structures. A powerful numerical capability and array handling facilities make it particularly useful in scientific applications. Finally, generic functions, user defined types, and operator overloading provide a basic introduction to object oriented programming.

The language is embedded in an environment that makes it simple for students who have no prior computing skills to use and to debug, and a number of books at various levels have been and are being published. One of these, [2], contains the complete language definition. The full BNF description of the language is available at the URL `http://www.imagine1.com/imagine1/`. Thus, we consider that F is now a real option for introduction as a first teaching language in many computing classes.

2 Easy and safe

A language intended for teaching must not only contain an adequate set of programming facilities, it must be safe and predictable. Also, the syntax should be unambiguous, allowing only one way to express any particular concept. We describe here these aspects, before turning to its powerful features in the following section.

In the F programming language, there are only two types of independently compilable program units: the main program and the module. Thus, apart from trivial problems that can be solved without recourse to other procedures, the bulk of a real program has to be split into one or more modules whose procedures are invoked by the main program. In this way, a modular style is not simply encouraged, but actually imposed on the student, who thereby acquires this important habit immediately. Of course, an advantage of this style is that all procedure interfaces are explicitly available at compile time, enabling a complete check of them by the compiler, leading to early detection of errors and a fast debugging cycle. In addition, F requires that each module entity be declared `public` or `private`, giving a high degree of protection to module variables and procedures.

F is a strongly typed language, requiring all typed entities – variables and functions – to be explicitly typed. Any entity detected by the compiler that has not been assigned a type, such as a misspelt variable name, will cause it to flag an error. Furthermore, any other attributes that an entity might have, for instance that it is dynamically allocatable at run time, must be specified on the associated type statement. In this way, all the attributes of a given entity are immediately visible.

The long-standing controversy over the `GO TO` statement has been resolved in F by excluding it, together with any form of statement label. The combined set of control structures in F – if-then-else, case, do with exit and cycle – ensure adequate control of program flow. For those situations where some form of exception handling is to be expected, particularly in the input/output and allocate statements, a testable return variable is made available.

Another safety measure is to require that all procedure arguments be specified with an `intent` attribute specifying whether the argument is an input, output, or input/output variable. Unintentional overwriting and the returning of undefined quantities thus become nearly impossible.

Finally, a degree of self-documentation is imposed, as each module or procedure is terminated with an end statement bearing its name, and the language keywords and the names of the intrinsic functions are reserved words, preventing them from being used in confusing ways.

To give the flavor, here is a module procedure to calculate the mean and variance of an array *x*:

```
subroutine calculate(x, mean, variance, ok)
   real, dimension(:), intent(in)  :: x
   real, intent(out)               :: mean, variance
   logical, intent(out)            :: ok
   integer                         :: n
   n = size(x)          ! size is an intrinsic function
   ok = n > 1
   select case (n)
   case (2:)
      mean = sum(x)/n   ! sum is an intrinsic function
      variance = sum((x-mean)**2)/(n-1)
   case (1)
      mean = x(1)
      variance = 0.0
   case default
      mean = 0.0
      variance = 0.0
   end select
end subroutine calculate
```

Taken together, the strict syntax requirements enable a student to advance quickly once the basic rules have been grasped. F is easy to teach, easy to learn, and easy to debug.

3 Modern features in F

F provides a complete array processing facility, including memory allocation and deallocation; a pointer facility; user defined types and operators on them; and a large number of intrinsic functions.

Arrays are first class objects and there is great flexibility in their use and dimensions. Arrays can be of any type, including user-defined types. Since defined types can have array components, it is possible to have arrays of arrays to any depth.

Arrays have a shape, which is the combination of rank (number of dimensions) and extents in each dimension. Extents can be variable; by default array subscripts start at one, but the starting and ending value can be declared with any values. The rank is a declared attribute and must be from 1 to 7. Further properties are:

- Arrays can have a constant shape in a main program, in a module, or in a module subprogram.
- Dummy argument arrays assume the shape of the actual argument used in the procedure invocation. The F compiler automatically passes on the shape as part of the call.
- Within a procedure, automatic arrays are local arrays that have a shape that is determined on entry. This is usually used to give working arrays a size based on the size of dummy arguments. For example:

```
subroutine  example (A, N)
   integer, intent(in)  ::  N
   real, intent(inout), dimension (:,:)  ::  A
   real, dimension (size(A,2), size(A,1), N)  ::  copies_of_A_transpose
```

creates a local array that can hold N copies of the transpose of the argument array A.

– Allocatable arrays can be allocated and deallocated at run-time and allow storage to be sized to run-time requirements. Unlike automatic arrays they can persist between procedure calls.

F's rich array processing facility goes further. All of the operators and essentially all of the intrinsic functions operate on arrays in an element-by-element way. If A, B, and C are arrays, then `A = B + sqrt(C)` is equivalent to executing the statement for corresponding elements of the arrays. The arrays in such an expression must be "conformable", that is, have the same shape, but they do not need to have the same subscript values as long as they have the same number of elements in each dimension. Scalars can be freely intermixed with arrays and "conform" in shape to any array.

Section subscripts access array elements in a regular pattern. If A is declared as a rank-two array, then `A` and `A(:,:)` both refer to the entire array, `A(1:10,1:M)` is a two-dimensional 10xM corner of the array, and `A(I:J:K,3)` is a one-dimensional section of the 3rd column of A whose elements are `A(I)`, `A(I+K)`, `A(I+2*K)` ... . The last element is usually `A(I+n*K)` where *n* is the largest integer such that I+n*K is less than or equal to J. If K is negative then the section runs backwards.

Sections, or even whole arrays, can be zero-sized, for example if M above is less than 1. In this case an operation on the array is essentially a "do nothing" and there is no need for explicit tests for end cases.

Vector subscripts access irregular patterns. If INDEX is an array of integers then `B(INDEX)` is the array with elements `B(INDEX(I))` for all valid subscripts I.

There is a `where` construct which masks array assignments.

```
where (A > 0)
   B = sqrt(A)
elsewhere
   A = 0
   B = 0
end where
```

calculates the square root where it makes sense and zeros out array elements elsewhere.

Pointers [1)] can be used to point to arrays or array sections. Although normally used for list processing they are also very useful to point to sections of arrays when the subscripts are tedious to write out. For example, if INSIDE is a pointer to a two-dimensional array, then

```
INSIDE => A(2:N-1, 2:M-1)
```

will point to the "inside" of the NxM array A.

As well as pointing to existing memory, pointer arrays can also be directly allocated, the normal way to create storage in nodes of linked lists.

The programmer must put the `target` attribute on any variable which might be pointed to. F compilers have a reasonable chance to optimize code because of this restriction.

F also supports user-defined data types, structures of such types, and user-defined operators for them. Operators, such as + or `sqrt`, can be overloaded and operate on user-defined types. Types can be recursive and contain elements of the same type to form lists and trees. Defined types can be "private" and the components are not directly available to the user – functions or defined operators must be used – or they can be "public" and the user can directly set and reference components. In the latter case there is also an automatically created constructor function for the type.

F supports all of the usual data types: real, complex, integer, logical, and characters and all of the operations on them that make sense. F gives special support to numerical precision and accuracy for the real and complex data types. They can be declared as ordinary "real" or "complex" and they get the default single precision on the given platform. However, using a set of intrinsic inquiry functions they can

---

[1)] Note: F pointers are not like C pointers – they are in the form of an attribute given to an entity, rather than being a data type in their own right – and so (dangerous) pointer arithmetic is not possible. This was a choice also made for Java. Dereferencing is automatic – no special symbol is required.

also be declared to have a specified minimum precision and/or exponent range. Thus, if the expression `selected_real_kind(10,50)` is used in a declaration sequence, the variables will have at least 10 digits of precision and an exponent range of at least 50. On some machines this is "single" precision and on others it is "double". It is a compile-time error to ask for more precision than the hardware can support.

F provides all of the common mathematical functions, sin, max, log, etc.; a built-in random number generator; a set of inquiry functions for the floating-point data types, epsilon, huge, tiny, etc.; and string manipulation functions that locate substrings within a string or trim trailing blanks, etc. It also has a complete set of array functions such as transpose or matrix multiplication and a set of mathematical reduction functions, sum, product, etc., as well as a set of logical reduction functions such as count which counts the number of true instances in its argument. The reduction functions can reduce an entire array of any rank to a scalar or they can reduce along a particular dimension, producing row sums or column sums for example.

We conclude this section with an example of a module procedure to locate a node of a given name in a tree structure.

```
! Look for name in the tree rooted at root. If found,
! make the module variable current point to the node
recursive subroutine look(root, name)
   use node_module ! definition of a node and access variable current
   type(node), intent(in), target :: root  ! user-defined type
   character(len=*), intent(in)   :: name
   type(node), pointer    :: child
!
   if (root%name == name) then             ! % qualifies a component
      current => root                      ! pointer assignment
   else
      child => root%child
      do
         if (.not.associated(child)) then  ! intrinsic function
            nullify(current)               ! null if name not found
            exit                           ! exits loop
         end if
         call look(child, name)
         if (associated(current)) then
            return
         end if
         child => child%sibling
      end do
   end if
end subroutine look
```

4 Teaching F

F is being used to teach the introductory one-semester programming course at Ventura College. The only prerequisite is college algebra. The course is taken primarily by sophomore students planning to transfer to one of the state Universities in engineering or science. This course fulfils the lower division programming requirement for these students.

The Windows 95 version of F ships with an environment, F_world, and the product installs very easily from the three supplied disks. The student version may be run from either a command line or from the F_world environment. The command-line option behaves as a classical compiler/linker, whereas F_world is an integrated environment combining editing, compiling/linking, and a language documentation facility.

The F_world interface is used for the beginning students. This requires the students to create each module as a separate window then validate each module in the correct order. F_world is straightforward, intuitive, and quickly mastered by all students. F diagnostics are rather good. The policy is that if a serious error is discovered early in the listing, a misspelt keyword for instance, that line is diagnosed and the scan is terminated. This scheme appears reasonable, and has been routinely adopted in teaching students in other languages – correct the first error, ignore the rest, and obtain another error scan.

The F_world environment supports a fill-in-the-blanks keyword template capability, but at Ventura College only the editing features of the environment are used. It supports multiple windows, and a reasonable full editing capability.

The command line options of F are not used by students, but are used often in lectures and for preparation Although this method might be preferred and is shown to students, they have already found a method that works (F_world environment, editor mode), and rarely change.

The actual purpose of this course is:

1. To give an introduction to computing for the engineer or scientist.
2. To teach those students how to solve elementary problems with the computer using F.
3. To generate in the students an appreciation for the power of the computer.
4. To do the above without generating computer phobia.

This is a first course in programming and, as is usual for this class, a few of the students have no prior computer experience, a few have Basic, and one was repeating the introductory course after dropping out of a previous Fortran 77 class taught by an unstructured engineer. This student's interesting comment was that F is a lot better as he had started in a class where extensive GOTOs and statement labels were used. After a couple of sessions with F's constructs, he really came to appreciate them. With F, all the objectives were met at least as well as has been done previously using Fortran or Pascal.

Overall, the class did well, retaining about the same percentage of the presented material as in a regular Fortran class. However, with F they have a good introduction to pointers, array operations, and modern control structures. F has been demonstrated to be a good introductory language that grows on the lecturer. It is small, consistent, and has no redundancies. Therefore, it's lean, not overinflated as are the languages designed by large committees with the history, legacy code, and compromises therein.

There are several features of F which are particularly appreciated in teaching to the beginner. The visibility of both variable and procedure names is simpler to define than in Pascal, and is a real improvement over older Fortrans. The concept of a 'statement' is clear and simple. There are no compound statements such as the if in Pascal. The only form of the comment is the ! – any text on any line after one ! in any position is comment. This makes it simpler for the student and simpler for the teacher – one never has the problem of the line of code being looked at actually being commented out by being embedded within some { } or /* */ over many lines.

Thus, F is to be preferred to Pascal as an introductory course for engineering students. They are exposed to a modern and structured method of writing programs, and do so using a language they are more likely to encounter in the real world of scientific/engineering simulations, analysis, and research

5 The Origins of F

Compilers for F are now available for Windows 95/NT, Windows 3.1/DOS, most UNIX machines, and the Macintosh PowerPC. How can F be available on so many platforms? F is actually a subset of Fortran 90 and F compilers are based on existing Fortran 90 compilers. When most people think of Fortran they think of [2)]

```
445 IF( ZONE(j4-1)) 470,485,455
450 k3= k3+1
    IF( D(j5)-(D(j5-1)*(T-D(j5-2))**2+(S-D(j5-3))**2 +
   1                    (R-D(j5-4))**2)) 445,480,440
455 m= m+1
    IF( m-ZONE(1) ) 465,465,460
460 m= 1
465 IF( i1-m) 410,480,410
```

However, Fortran underwent a significant modernization in 1990 when many new features were added. Fortunately for existing codes (and coders), the features were added to the language. Unfortunately for

2) This is an excerpt from the Livermore Loops, an important set of performance measurement codes. For non-Fortran people, the 3-branch IF would transfer control to the first statement label if the test expression was negative, to the second label if the expression was zero, and to the third label if the expression evaluated to greater than zero. Production quality optimizing compilers could "easily understand" this particular sequence in the mid '80s. Production quality humans still have trouble with it.

new programmers (and new programs) nothing was removed from the language. The F subset removes all of the old, redundant, confusing, difficult-to-use syntax and semantics and retains the new modern features, and does it in a way that preserves access to existing code and to Fortran's historic emphasis on high performance computing. Since it is a subset of a widely used language, a student who learns F won't have to unlearn anything after graduation; they might have to learn a little more, but that isn't all bad.

In some engineering curricula, Fortran is the first and only language required of majors. Fortran 90 has now become a big, cumbersome language supporting many obsolete features, yet the beginning engineer/scientist needs to learn it. F offers a structured introduction to that language which the engineer/scientist will use to analyze systems and create his own programs, even large ones. Since the only real pedagogical concern is that the student does not see Fortran as it actually exists in the real world, a little time at the end of a course might be devoted to discussing the obsolete features they may expect to see there.

But the thrust of F is not just as an entry into the world of Fortran – it is intended as a valid replacement for Pascal, Basic, and especially C as a first programming language.

6 Availability of F

F is currently available from Imagine1 at `http://www.imagine1.com/imagine1/` or, by e-mail, at `info@imagine1.com`. There are also several F programming books including over 300 example programs available from the Web site.

7 References

[1] Trott, P., Programming Languages, Past Present and Future, *ACM SIGPLAN Notices*, 32(1), pp. 14-57, 1997.

[2] Metcalf, M. and Reid, J., *The F programming language*, (Oxford U. Press), 1996.