Fortran 95 for Fortran 77 users

J.F.Harper
School of Mathematics, Statistics and Computer Science
Victoria University of Wellington
First ed. 5 Apr 2006, latest amendments 23 May 2007

1 Introduction

With its first compiler delivered in April 1957, Fortran has been for 50 years a computer language used mainly by engineers and scientists (but by few computer scientists), mainly for numerical work. Five of its versions have been standardised and are commonly referred to as f66, f77, f90, f95 and f2003 to indicate the year. F95 appears to have superseded f90, and no f2003 compilers exist yet, so these notes concentrate on f77 and f95.

Various f95 compilers exist, some of them free. Some of the new features make programming easier, some allow the machine to detect bugs that f77 compilers cannot, and some make programs easier to read.

The "learning curve" for f77 users wishing to start using f95 is not very steep, because one may use as many or as few of the new features as one wishes to. It is, however, long: the f95 standard is about twice as long as the f77 one.

These notes are written for f77 users, to describe a number of the f95 features that I found so useful that I gave up f77 except when writing a program for someone who had no f95 access, or when testing the differences between various Fortran dialects. I begin by recommending some books and compilers.

2 Recommended books and a Web page

On f77: Michael Metcalf *Effective Fortran 77*, Oxford (1990)

On f90 and f95: Michael Metcalf and John Reid *Fortran 90/95 Explained*, Oxford (1999)

On f95 and f2003: Michael Metcalf, John Reid and Michael Cohen *Fortran 95/2003 Explained*, Oxford (2004), and this Web page by Metcalf:
`http://en.wikipedia.org/wiki/Fortran_language_features`

All the above are as concise as the subjects permit, and are comprehensive. In Metcalf et al.(2004), Chapters 1–10 are mainly on f95, Chapters 11–19 are mainly on f2003, and Chapter 20 is mainly on superseded f77 features that may still be used.

The following is also comprehensive, less concise, but on some matters is better as a reference book:

Jeanne C. Adams et al. *Fortran 95 Handbook*, MIT Press (1997).

There are also the Fortran standards. These are sold for high prices, but Google may help you find a free one. Many Fortran programs contain floating-point arithmetic. That is convenient but it has traps for young players. Googling "goldberg floating point" will help explain why.

There is a newsgroup called comp.lang.fortran for discussions on the language; its signal-to-noise ratio is higher than most, and some real experts frequently contribute (e.g. members of the committee revising the Fortran standard, a Fortran textbook writer, and at least two senior members of major computer companies' Fortran teams.)

2.1 Recommended compilers and bug-hunting

Good free compilers, available on many kinds of machine, are `g77` for Fortran 77 and `g95` (which I find better than `gfortran`) for Fortran 95 with a few Fortran 2003 features; The ITS machine mahoe has `g77`, the Compaq compiler `f95`, and the NAG compiler `nagf95`. SMSCS Sun machines have `g77`, `g95` and Sun `f95`. I have used all of these, and I recommend trying more than one compiler if you don't understand or don't believe an error message.

If you get different *run-time* results from two compilers (such as one giving what seems to be good output and another crashing with a core dump) you have probably committed one of the many errors that compilers are not required to diagnose. If your program causes a *compile-time* core dump, you have found a compiler bug that ought to be reported, whether your own program is correct Fortran or not. I have found more bugs in g95 than in any other compiler, but they get fixed faster when reported. When writing this in April 2006 I knew of only one unfixed g95 bug, but of three unfixed bugs in another compiler.

3 Upper and lower case

Strict f77 required everything to be in UPPER CASE, but most of us find it easier to read a mixture of upper and lower. Many f77 compilers allowed both, and in f95 it's always allowed, with upper and lower treated as equivalent: `PRINT`, `print` and `PrInT` all say the same thing to the compiler. Some people like to put keywords such as `IF`, `PRINT`, `DO` in upper case, and everything else in lower case; others use lower case for everything.

4 Quotation marks

In f77 character strings had to begin and end with apostrophes (') so an apostrophe inside a string had to be two together (''). In f95 you may do that, or you may begin and end with quotation marks ("). These two statements print the same thing:

```
PRINT '(A)',' Student''s t test'! OK in f77 or f95
PRINT '(A)'," Student's t test" ! OK in f95
```

The f66 method (`17H Student's t test`) was already obsolete in f77, though some compilers still let you use it.

5 Underscores in names

In f95 the underscore (_) may be used as well as letters or digits in a name of a program entity, but the first character must still be a letter. Names may now be up to 31 characters long, instead of the official f77 limit of 6.

6 Semicolons separating statements

In f95 a semicolon (;) ends a statement, so you may put more than one statement on a line, e.g.

```
i = 1; j = 2; k = 3
```

This allows you to see more of your program in one computer screen but your readers may fail to notice the later statements, and so misinterpret what they did read. Many people avoid the f95 semicolon for that reason.

7 Comments using !

In f95 an exclamation mark (!) anywhere, except space 6 on a fixed source form line (see Section 9 below), means that the rest of that line is comment for human readers, and will be ignored by the compiler. Example:

```
x = sqrt(y) ! but what if y is negative ??
```

8 Relational operators: .LT. etc.

In f77 there were six logical operators between numeric or character expressions: `.LT.` `.LE.` `.EQ.` `.NE.` `.GT.` `.GE.` They often occurred after `IF`, e.g.

```
IF (x.LT.y) STOP
```

In f95, you may (but need not) replace them by symbols more like what mathematicians use:

`.LT.`	is equivalent to	`<`
`.LE.`	is equivalent to	`<=`
`.EQ.`	is equivalent to	`==`
`.NE.`	is equivalent to	`/=`
`.GT.`	is equivalent to	`>`
`.GE.`	is equivalent to	`>=`

The logical operators (`.NOT. .AND. .OR. .EQV. .NEQV.`) are unchanged from what they were in f77. Warning: f95 also has `=>` but it's nothing to do with these operators. See Section 20.

9 Free source form

In f77 various things had to be in particular places on the line: comments started with `*` or `C` in space 1, statement numbers had to be in spaces 1-5, anything except 0 or blank in space 6 indicated that that line was continuing a statement from the previous line, actual Fortran code had to be in spaces 7-72, and spaces were irrelevant except in character strings such as `'Hello World'`. That is called fixed source form.

Free source form avoids much of that hassle. In it, `C` and `*` do not mark comments; `!` does. Continuation of statements is marked with `&` at the end of the *previous* line instead of something in space 6 on the *next* line, statement numbers (if used) must be the first nonblank thing on a line, actual Fortran code may go on to space 132, and keywords, variable names and some other things must be separated by one or more blank spaces. So, if the `&` on line 3 of this program is in space 73, and the `+` on line 4 is in space 6:

```
      INTEGER j,k,jk
      j = 1; k = 2; jk = 666
      PRINT *, j                                                    &
     + k
      END
```

then the program is valid f95 in either source form, but it will print 666 if compiled as fixed source form, 3 as free source form.

Many compilers assume that a program file whose name ends in `.f90` is in free source form, and one whose name ends in `.f` or `.for` is in fixed source form. There are usually options called something like `-fixed` or `-free` (but `-ffixed-form` or `-ffree-form` in g95) to change that.

10 IMPLICIT NONE

If you put `IMPLICIT NONE` at the top of your program you must declare every variable, instead of everything beginning with I,J,K,L,M or N being assumed to be a scalar integer, and everything else scalar "real" (single-precision floating point), unless you specified otherwise. `IMPLICIT NONE` is a very good way to catch hard-to-see bugs, e.g. without it the notorious

```
      DO 666 i=1.9
```

in a fixed source form program makes the machine put a variable `DO666I` equal to 1.9, but with it, you will probably be told you hadn't declared `DO666I`, which is likely to be useful information. It won't tell you the trouble was writing 1.9 instead of 1,9 but you will be led to the right place.

11 Loop features

11.1 DO and END DO, and naming loops

Instead of

```
      DO 666 ...
          ...
 666  CONTINUE
```

you may now write

```
      DO ...
         ...
      END DO
```

And you won't have to look for `GOTO 666` elsewhere to find out what the program might do. You can still make it clear which `DO` goes with which `END DO` by giving your loop a "construct name" such as `bigloop` below:

```
      bigloop: DO ...
         ...
      END DO bigloop
```

11.2 Integer-only index

If `X` is a real (floating-point) variable, then

```
      DO 666 X = 0.0, 1.0, 2.0/3000
```

is a valid beginning of a DO loop in f77, but it's inadvisable because the vagaries of floating-point arithmetic imply that the last value of `X` may be over 1.0. When I tried it with the g77 compiler the last `X` was 1.00001168. In f95 the loop index (if any: see Section 11.3 below) must be an integer,

which prevents that sort of trouble. Using a non-integer loop index is one of the very few ways in which a valid f77 program may become invalid in f95.

11.3 CYCLE, EXIT, and no loop index

On executing a `CYCLE` statement in an f95 loop, the rest of the loop is ignored that time round. On executing an `EXIT` statement, that loop immediately finishes. In this example, `stuff1`, `stuff2`, `stuff3` represent Fortran statements (possibly several of them). They may change the value of `j`, but changing `i` inside the loop is forbidden:

```
 DO i = 1,1000
    stuff1
    IF (j>10) CYCLE
    stuff2
    IF (j>20) EXIT
    stuff3
 END DO
```

As usual, on the first time round this loop `i` = 1, the next time `i` = 2, and so on until `i` = 1000 unless something happens to stop the loop before then. Each time round the loop, `stuff1` is executed. If `j>10` after that then `stuff2` and `stuff3` are ignored, and unless `i` was already 1000, `i` increases by 1, `stuff1` is executed again and the value of `x` is tested again.

If `j>20` just after `stuff2` is executed, then the loop ends immediately.

If the loop has a name (Section 11.1) you may put it after `CYCLE` or `EXIT` just as you do after `END DO`.

In f95 you may also have a loop with no "index" variable to be incremented! Example, in which `stuff` presumably changes the value of `x`:

```
DO
   stuff
   IF (x>20.0) EXIT
END DO
```

11.4 Array assignment

If your loop was merely going to do some calculations on arrays, you may not need a DO loop in f95. Examples, if `x, y` are arrays with the same bounds:

```
 y = x+2          ! adds 2 to each element of x,
 y = x*x          ! squares each element of x.
 x(3:4) = x(1:2) ! equivalent to x(3)=x(1); x(4)=x(2)
```

11.5 Elemental functions

Suppose you have declared `REAL x(n),y(n)`, where `n` is a constant, and you have evaluated `x`. The loop

```
DO i = 1,n
   y(i) = exp(x(i))
END DO
```

which sets each element of `y` equal to `exp`(the corresponding element of `x`) may be replaced by

```
y = exp(x)
```

as `exp` is one of the many standard functions that are "elemental" in f95, i.e. they may be applied to each element of an array. You may invent your own, but they must obey various restrictions: see the f95 books.

11.6 FORALL

Suppose you have declared `REAL a(n,n), x(n)` and evaluated `x`. Then

```
FORALL(i=1:n) a(i,i) = x(i)**2
```

sets the diagonal elements of the `n`×`n` matrix `a` equal to the squares of the corresponding elements of `x`. That would otherwise need a DO loop. To do several array operations together, there is the "FORALL construct":

```
FORALL (i=1:n)
  ...
END FORALL
```

which has the same relation to the FORALL statement that the f77 and f95 "IF construct"

```
IF (x>y) THEN
  ...
END IF
```

has to the IF statement

```
IF (x>y) ...
```

12 Initialisation and attributes in declarations

In f77 and earlier you often needed two or more separate statements for declaring something, telling the machine more about its properties, and initialising it, e.g.

```
      REAL PI
      PARAMETER (PI = 3.1415927)
```

In f95 they may be combined if you include `::` as shown below, to display

all the properties of PI on one line, e.g.

```
REAL,PARAMETER:: pi = 3.1415927
```

The word PARAMETER says that pi is a constant, and it cannot be changed elsewhere in the program. You may also initialise variables and/or specify their properties while declaring them; examples are

```
REAL :: x(3) = (/ 1.0, 2.0, 4.0 /)
REAL,DIMENSION(100,3) :: a,b,c,d,e,f,g
```

The catch with this sort of thing is that the RHS must be something that can be evaluated at compile time. In f95 that excludes functions of type real or complex, but in f2003 those are allowed. Example:

```
REAL::pi = acos(-1.0) ! OK in f2003 but not f95
```

(Note: blank spaces are optional before and after ::)

A whole one-dimensional array may be given a value with (/ /) around the list of elements, or in f2003 with [ ] as an alternative to (/ /). The list may contain an "implied-do" like those that could appear in DATA, READ, PRINT and WRITE statements in f77, e.g. if y is an array with 10 elements,

```
y = (/ (i**2,i=1,10) /)
```

will give y the values 1.0, 4.0, 9.0, ..., 100.0. That can be an executable statement anywhere in the program, or in the declaration of the array y.

13 INCLUDE

In f77 there was no standard way to include material from another file in your program. Various compilers had various ways to do it. In f95 there is a standard way:

```
INCLUDE 'foobar' ! or INCLUDE "foobar"
```

if foobar is the name of the file you want to include in your program. Warning: don't try to mix free and fixed source form!

14 INTENT

When declaring a subroutine or function, you may specify INTENT(IN) for arguments that are input only, INTENT(OUT) for those that are output only, an INTENT(INOUT) for those which are both. This lets both the compiler and the human reader check for various possible bugs. Example:

```
REAL FUNCTION cuberoot(t)
   REAL,INTENT(IN)::   t
   cuberoot = sign(1.0,t)*abs(t)**(1.0/3.0)
END FUNCTION cuberoot
```

That also illustrates another f95 aid to human readers: putting the word `FUNCTION` or `SUBROUTINE` followed by the name after its `END`.

15 ALLOCATE

In f77 any array in a main program had to have fixed bounds, e.g. in

```
      REAL x(n),y(n,n)
```

`n` had to be a constant unless `x` and `y` were dummy arguments of a subprogram, so you had to guess what value of `n` was the biggest you might ever need. In f95 you may instead declare `x` and `y` as

```
 REAL,ALLOCATABLE:: x(:),y(:,:)
```

and after `n`, which may be a variable, has been given a value, you may say

```
 ALLOCATE(x(n),y(n,n))
```

If you wish to change the size of an allocated array, you must `deallocate` it first. There is a standard function `allocated` to tell whether it has been allocated or not, so you may find yourself writing

```
 IF (allocated(x)) deallocate(x)
 allocate(x(2*n))
```

If you do that, `x` will now have twice as many elements as it used to, and the previous values of its elements will have been lost.

16 New standard ("intrinsic") functions

There are many of these; I give only a few examples.

16.1 MERGE

Among the many elemental functions (see Section 11.5 above) new in f95 is `merge(truestuff,falsestuff,mask)` in which `truestuff` may be of any type, `falsestuff` is of the same type, and `mask` is of type logical. If `truestuff` is an array, `falsestuff` and `mask` must be arrays of the same size and shape. If `truestuff` is of type character, `falsestuff` must be the same length. The function returns `truestuff` if `mask` is true, `falsestuff` otherwise. Example:

```
 delta = merge(1, 0, i==j)
```

sets `delta` to $\boldsymbol{\delta}_{\mathtt{ij}}$: 1 if `i,j` are equal, 0 otherwise. Warning: Both `truestuff` and `falsestuff` may be calculated even though only one of them will be needed, so don't try to rewrite the `cuberoot` function in Section 14 as

```
 cuberoot = merge(0.0,merge(t**third, -((-t)**third),t>0),t=0)
```

16.2 KIND and SELECTED_REAL_KIND

Some old programs had declarations like `REAL*8 x`. That was never standard f77, and was always non-portable: some compilers use different numbers for a given precision. `REAL(8) x` is standard f95, but is still non-portable.

`Kind(t)` gives you the number that replaces that `8` for a variable of the same kind as `t`, e.g. `Kind(1d0)` for double precision. I use and recommend things like

```
INTEGER,PARAMETER:: dp = kind(1d0)
REAL(dp) x
COMPLEX(dp) z
```

Complex double precision was an extension to the f77 standard, but it's always available in f95. Some f95 compilers also provide quadruple precision. If they do, they must provide it for both complex and real numbers.

`Selected_real_kind` allows you to try to declare real or complex things with at least a specified number of significant digits, e.g.

```
INTEGER,PARAMETER:: dp = selected_real_kind(14)
```

gives the lowest precision with at least 14 digits, which is double precision in many compilers, single in some. If you ask for too many digits then `dp` will be negative, and declarations using it, like `REAL(dp) x`, will fail.

16.3 HUGE, EPSILON, TINY and PRECISION

If `x` is of any real or integer kind, `huge(x)` is the largest positive number available for that kind.

If `x` is of any real kind, `epsilon(x)` is the smallest positive number such that `1+epsilon(x)` is distinguishable from `1`, and `tiny(x)` is the smallest positive number available for that kind. It is *much* smaller than `epsilon(x)`.

If `x` is of any real or complex kind, `precision(x)` is the number of decimal significant figures you can expect for `x`, so `precision(1.0)` is commonly 6, `precision(1d0)` is commonly 15.

All these make it easier to check your numerical analysis. There are several more such "inquiry functions": see the books.

For example, g95 currently has the first two of these real kinds, and Sun f95 has all three:

	`precision`	`huge`	`epsilon`	`tiny`
"single precision"	6	3.4E+38	1.2E-7	1.2E-38
"double precision"	15	1.8E+308	2.2E-16	2.2E-308
"quadruple precision"	33	1.2E+4932	1.9E-34	3.4E-4932

The `precision` values are exact, the others are correct to the figures given.

16.4 SIZE, LBOUND and UBOUND

If `x` is an array, then `size(x)` is its total number of elements, and `size(x,n)` is the number of elements along dimension `n`. Example: if `x` was declared as

```
INTEGER x(3, 0:9)
```

then `size(x,1)` is 3, `size(x,2)` is 10, and `size(x)` is 30.

The function `lbound(x)` is the array `(/1,0/)` of lower bounds of the subscripts of `x`, and `lbound(x,2)` is the scalar `1`. In the same way `ubound(x)` finds upper bounds: `ubound(x)` is `(/3,9/)`, `ubound(x,2)` is `9`.

16.5 MAXVAL, MINVAL, MAXLOC and MINLOC

If `x` is the same array as in Section 16.4 above, then `maxval(x)` is its largest element and `maxloc(x)` is the array of subscripts at which that largest element is found. If there are two equal largest elements `maxloc` picks the first one in array-element order. If `y` is a one-dimensional array `maxloc(y)` is a scalar. `Minval` and `minloc` do the corresponding things for least elements. All four functions work on integer or real arrays (of any kind).

16.6 LEN, TRIM and LEN_TRIM

If `x` is a character constant or variable, `len(x)` is its length (as in f77), `trim(x)` is its value with all trailing blanks removed, and the length of `trim(x)` is `len_trim(x)`. Example: this program

```
CHARACTER:: x*8 = 'Hello'
PRINT *,'len(x)     =',len(x)     ,'      x ="',x,'"'
PRINT *,'len_trim(x)=',len_trim(x),' trim(x)="',trim(x),'"'
```

prints this output:

```
len(x)     = 8       x ="Hello   "
len_trim(x)= 5  trim(x)="Hello"
```

16.7 SUM, PRODUCT and MATMUL

If `x` is a real, integer or complex array, `sum(x)` is the sum of all its elements. There is also a `sum(x,n)` possibility, which sums only over subscript number `n` and returns an array with one fewer dimension than `x`. `Product` multiplies elements instead of adding them. `Matmul(a,b)` calculates the matrix product if one of `a,b` is two-dimensional, the other is either one- or two-dimensional, and the matrix product exists.

17 CONTAINS and explicit interfaces

Subroutines, and all functions except statement functions, are subprograms. In f77 they had to be outside your main program; they were usually written after its end in the same file, or in separate files and compiled separately. One consequence was the need for `COMMON` blocks in both the main program and the subprogram to make things from one available in the other without being arguments of the subprogram. Another consequence was that mismatches of type (e.g. integer in one place, real in the other) were not detected by the machine: you got wrong answers or core dumps. In f95 you may have a `CONTAINS` statement before the `END` of a main program (or of a module: see Section 20 below), and declare subprograms between `CONTAINS` and that `END`. If you do, you must end each subprogram with `END FUNCTION` or `END SUBROUTINE` and it's a good idea to put the subprogram name there too. A "contained" subprogram such as `cuberoot` below may use anything declared in its "container", such as `third`:

```
PROGRAM cube_roots
  IMPLICIT NONE
  INTEGER:: i
  REAL   :: x(5) = (/-2,-1,0,1,2/), third = 1.0/3.0
  PRINT '(F10.7,1X,F10.7)',(x(i),cuberoot(x(i)),i = 1,5)
CONTAINS
  REAL FUNCTION cuberoot(t)
    REAL,INTENT(IN)::    t
    cuberoot = sign(1.0,t)*abs(t)**third
  END FUNCTION cuberoot
END PROGRAM cube_roots
```

The compiler then automatically gives these subprograms what are known as "explicit interfaces". Subprograms that don't follow `CONTAINS` normally have "implicit interfaces", but you may write explicit ones for them, and you sometimes have to. Metcalf et al. or Adams et al. tell you when you must, and how to do it. `CONTAINS` avoids the hassle.

18 Arrays in functions/subroutines

In f77, arrays which were dummy arguments of subprograms could be specified with their last dimension `*`, e.g. `REAL x(*), y(3,*)`. Such an "assumed-size" array then had the same size as the actual array in the subprogram call. In f95 this can be extended to any or all dimensions by using a colon (`:`) instead of `*`, e.g.

```
 REAL c(0:), d(:,:) ! d has lower subscript bounds 1, as usual
```
The subprogram must then have an explicit interface (see Section 17).

Arrays such as `c` and `d` are called "assumed shape". Besides allowing array subscripts other than the last to be possibly different each time the subprogram is used, your compiler can do various useful things with the bounds, and can check (if you ask it nicely) whether you have gone outside the declared bounds of the array. It probably doesn't check automatically because that makes programs run slower. Example:
```
 INTEGER:: i2(5)=(/ 2,4,6,8,10 /)
 PRINT '(5I4)',i2
 PRINT '(5I4)',square(i2)
 CONTAINS
   FUNCTION square(       n)
     INTEGER,INTENT(IN):: n(:)
     INTEGER  square(size(n))
     square = n*n
   END FUNCTION square
 END
```
This prints
```
   2   4   6   8  10
   4  16  36  64 100
```

Here `square(n)` was declared to be a function returning an array the same size as its input array `n`; you couldn't use `n(*)` instead of `n(:)` when declaring it because the compiler wouldn't be able to tell what `size(n)` was. Functions that return arrays instead of scalars didn't exist in f77; they need explicit interfaces (Section 17).

As well as assumed-size arrays (with `*` as the last bound) and assumed-shape arrays (with `:` among the bounds), a subprogram may contain yet another kind: the "automatic" array. This is an array which isn't itself an argument of the subprogram, but which has bounds depending on those arguments. That's OK in f95 but not in f77. Metcalf et al. (2004) gave an example:
```
 SUBROUTINE swap(a,b)
   REAL, DIMENSION(:), INTENT(INOUT):: a,b
   REAL, DIMENSION(size(a)):: work
   work = a
   a = b
   b = work
 END SUBROUTINE swap
```
in which the automatic array `work` is used to interchange the contents of two assumed-shape arrays `a,b`.

19 RECURSION

In f95 a function or subroutine may call itself if you specify `RECURSIVE` when declaring it and give it a `RESULT` clause. Mathematicans are fond of illustrating this with factorials. Even though there are better ways to calculate factorials, I follow their tradition here:

```
PRINT *, 'This gives factorial(n) for n>=0, -1 otherwise.'
PRINT "(2(A,I3))", (' n =',n,' fac(n) =',fac(n),n = -1,4)
CONTAINS
  INTEGER RECURSIVE FUNCTION fac(n) RESULT(answer)
    INTEGER, INTENT(IN) ::       n
    IF (n<=0) THEN
       answer = sign(1,n)
    ELSE
       answer = fac(n-1) * n
    END IF
  END FUNCTION fac
END
```

It prints

```
This gives factorial(n) for n>=0, -1 otherwise.
n = -1 fac(n) = -1
n =  0 fac(n) =  1
n =  1 fac(n) =  1
n =  2 fac(n) =  2
n =  3 fac(n) =  6
n =  4 fac(n) = 24
```

A recursive function may not be elemental (see Section 11.5).

20 MODULE

You may put declarations of constants, variables and subprograms between `MODULE modname` and `END MODULE modname`. That declares a "module" called `modname`. If there are subprograms they must go between `CONTAINS` and `END MODULE` just as subprograms inside a main program had to go between `CONTAINS` and `END PROGRAM`. There are three main reasons for writing modules.

1. The same module may be used in several different main programs, subprograms or other modules, and one main program, subprogram, or module may use several different modules.

2. Modules make most `COMMON` and `BLOCK DATA` statements superfluous. They were always bug-prone.

3. Compilers will detect many bugs due to mismatches between what's declared in a module and how it's used that they can't detect otherwise.

To use a module in a program or in another module, write

```
USE modname
```

at the beginning, after any `PROGRAM`, `FUNCTION`, `SUBROUTINE` or `MODULE` statement but before anything else. That makes available what's inside the module, including subprograms. The interfaces of subprograms in modules are automatically explicit (see Section 17). Things in a module may be declared `PRIVATE` and then they can be referred to from within the module but not from outside.

You may wish to use only some of the public (non-`PRIVATE`) things (say `x, y`) from a module. If so, say

```
USE modname, ONLY: x, y
```

Many people always put `ONLY` clauses in their `USE` ststements because they document in the "using" program what was actually used.

You may also change the name in your program of a variable from the module. For example, if `x` in your program is one of your own variables, and you also want to use the `x` in the module, you could rename the module's `x` as (say) `xmod` in your program by saying

```
USE modname, ONLY: xmod => x, y
```

if you also want an `ONLY` clause, or

```
USE modname, xmod => x
```

if you don't.

Note: `=>` is also used in "pointer assignment", a topic I don't want to get into here. F95 allows for pointers and their targets, but allocatable arrays can do some of the things that pointers can, and I haven't yet needed to do one of the jobs that only pointers can do.

Conclusion

These notes have merely scratched the surface of f95: after all they occupy only 15 pages, while the books on f95 all have several hundred pages.