Modern Fortran for Fortran 77 users

J. F. Harper
School of Mathematics and Statistics
Victoria University of Wellington, New Zealand
This version 3 Nov 2021

Contents

1 Introduction

With its first compiler delivered in April 1957, Fortran has been for over 60 years a computer language used mainly by engineers and scientists (but by few computer scientists), mainly for numerical work. Every few years a new Fortran standard appears. They are commonly referred to as f66, f77, f90, f95, f2003, f2008 and f2018 to indicate the year. F95 has completely superseded f90, and some compilers can't cope with the whole of f2003 yet, while others support parts of both it and f2008, and even f2018.

These notes are written for f77 users, to describe a number of the f95 features that I found so useful that I have given up f77 except when checking the differences between f77 and f95. They merely scratch the surface of f95: after all they occupy only 22 pages, while the books on f95 all have several hundred pages. A few post-f95 features are also mentioned.

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.

Some things that used to be valid Fortran are now "obsolete" (no longer in the standard, but often still available with many compilers) or "obsolescent" (still valid Fortran but may become obsolete in a later version).

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

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 later: Michael Metcalf, John Reid and Michael Cohen *Fortran 95/2003 Explained*, Oxford (2004) or *Modern Fortran Explained*, Oxford (2011, 2018) and this Web page by Metcalf:
`http://en.wikipedia.org/wiki/Fortran_language_features`

Jeanne C. Adams et al. *Fortran 95 Handbook*, MIT Press (1997), and *The Fortran 2003 Handbook*, Springer (2009).

I have used all those except Metcalf et al. (2011). I find Metcalf's books better as textbooks, Adams et al. better as a reference book. 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 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. past or present members of the committee revising the Fortran standard, a Fortran textbook writer, and 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` and `gfortran` for Fortran 95 with later features; you may also qualify for free use of `ifort` or the Oracle (formerly Sun) compiler `f95`. You can download your own `g95` and `gfortran`. Beware: `gfortran` occupies a lot of disk space and takes over an hour to install if you start with its source code.

Try two or more compilers if you don't understand or don't believe an error message. If two compilers give diffent *run-time* results (especially one giving what seems to be good output and another crashing) you may have committed one of the many errors that compilers are not required to diagnose, or you may have used an f2003 or f2008 or f2018 feature that one of the compilers doesn't yet have. If your program causes a *compile-time* internal error, you have found a compiler bug that ought to be reported, whether your own program is correct Fortran or not.

Although g95 offers quadruple precision on some machines (see §17.3 on page 13) it does not yet have some of the standard functions, e.g. log, sin, cos, in that precision. Work on it seems to have stopped in February 2013; my web site `http://homepages.ecs.vuw.ac.nz/~harper` reports bugs I have found since, at the line beginning

`. . . . Six unfixed bugs in g95`

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 (now including myself) use lower case for almost everything in a program unless it's f77.

4 Free and fixed source form, and program names

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. You may still use it but it is obsolescent.

Free source form avoids the space-counting 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. Variable names, many keywords, and some other things must be separated by punctuation or one or more blank spaces, but must not include blanks, e.g. `endif` and `end if` are OK but not `en dif`.

You may need to write programs valid in both source forms. The f2003 standard recommends putting `&` in space 73 of a line which contains a statement to be continued on the next line, and another `&` in space 6 of that next line. Unfortunately it also says that if a fixed source form line contains only default kind characters then it shall contain exactly 72 characters. Fortunately, compilers usually ignore that requirement. Some used to require 80.

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. Don't end the name with `.f95` or `.f03` etc. because some compilers disallow that. There are usually options called something like `-fixed` or `-free` (but `-ffixed-form` or `-ffree-form` in g95) to change the source form.

5 Comments using !

In f95 an exclamation mark (`!`) anywhere, except space 6 on a fixed source form line (see §4 on page 5), 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 ??
```

The f77 comment marker (`*` or `C` in space 1 on the line) is still available, but only in fixed source form.

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

7 Underscores in and max. length of 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.

8 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 won't use the f95 semicolon for that reason.

9 Relational operators: .LT. or < 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 write them in lower case, or replace them as follows:

`<`	means the same as	`.LT.`
`<=`	means the same as	`.LE.`
`==`	means the same as	`.EQ.`
`/=`	means the same as	`.NE.`
`>`	means the same as	`.GT.`
`>=`	means the same as	`.GE.`

The logical operators (`.NOT. .AND. .OR. .EQV. .NEQV.`) are still the same as in f77. Note: f95 also has `=>` but it does not mean "implies". See §21 on page 17 for one of its uses.

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 integer, and everything else real, 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 gives a variable called `DO666I` the value 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 it will show you where to look in your program.

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. Don't end the name with `.f95` or `.f03` etc. There are usually options called something like `-fixed` or `-free` (but `-ffixed-form` or `-ffree-form` in g95) to change that.

11 New features of DO loops

11.1 Do-variables must now be integers

In f77, if `X` is a scalar (non-array) real variable, then it may be used as a "do-variable" automatically incremented each time around a loop, e.g.

```
      DO 666 X = 0.0, 1.0, 1.0/1500.0 ! Good f77, bad f95
```

It's not a good idea because the vagaries of floating-point arithmetic allow the last value of `X` to be over 1.0. When I tried it with the g77 compiler, the last `X` was 1.00001168. In f95 the do-variable (if any: see §11.4 on page 9) must be an integer, which prevents that sort of trouble. Using a non-integer do-variable is one of the very few ways in which a valid f77 program may become invalid in f95. Nevertheless some f95 compilers allow it.

11.2 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 (let alone `ERR=666` or `END=666`) to find out how else the program might reach that `CONTINUE`.

You can still make it clear which `do` goes with which `end do` by indenting, or by giving your loop a "construct name" such as `bigloop` below:

```
bigloop: do ...
   ...
end do bigloop
```

If you name your loop you must put its name after its `end do`.

WARNING: the largest default integer is `huge(1)`. With this loop using it:

```
do i = huge(1)-1,huge(1)
  print *,i
  if(i<0)exit
end do
```

some compilers using 32-bit integers will print

```
2147483646
2147483647
-2147483648
```

Other compilers just print the first two of those lines. See §22 on page 18 to understand why. Also, see §11.3 on page 8 to explain `exit`.

11.3 CYCLE and EXIT

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 NOT the value of `i`:

```
do i = 1,1000
   stuff1
   if (j > 10) cycle
   stuff2
   if (j > 20) exit
   stuff3
end do
```

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

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

If the loop has a name (§11.2 on page 7) you may (but are not required to) put it after `cycle` or `exit`; you must put it after `end do`.

11.4 Loops with no do-variable

In f95 a do loop need not have a do-variable. It will then go on until something stops it. In this example the statement(s) represented by `stuff` had better change the value of `x`:

```
do
   stuff
   if (x>20.0) exit
end do
```

12 Avoiding DO loops

12.1 Array assignment, (/ /) and [ ]

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

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

You can also specify a whole one-dimensional array, or array section like `x(3:4)`, by putting its list of values between (/ and /), or in f2003 between [and]. The sort of "implied do" that f77 allowed in DATA statements or input/output lists is also OK in these "array constructors", e.g.

```
 x = (/ (i**2,i=1,size(x)) /) ! values 1,4,9,...
```

The RHS is supposed to be evaluated before subs. into LHS. If `a` is `[1,2]`, then `a = [a(2),a(1)+a(2)]` should give `[2,3]` but a g95 bug gives `[2,4]`.

12.2 Elemental functions

Suppose `x` and `y` are arrays declared `real x(n),y(n)` and `x` is known. Then

```
 do i = 1,n
    y(i) = exp(x(i))
 end do
```

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

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

as `exp` is one of the many standard functions that are now "elemental", i.e. they can apply to each element of an array. You may invent your own elemental functions, but there are various restrictions: see the books.

12.3 FORALL

Another way to avoid DO loops: suppose you declared `real a(n,n), x(n)` and you have 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`. There is also the "FORALL construct":

```
forall (i=1:n)
  ...
end forall
```

It has the same relation to the FORALL statement that the "if construct"

```
if (x>y) then
  ...
end if
```

has to the if statement

```
if (x>y) ...
```

FORALL was made obsolescent in f2018 but you may still use it.

13 New features of declarations

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

```
      DOUBLE PRECISION E
      PARAMETER (E = 2.7182818284590452D0)
```

In f95 they may be combined if you include `::` as shown below, to display all the properties of `e` on one line, e.g.

```
double precision, parameter:: e = 2.7182818284590452d0
```

As in f77, `parameter` says that `e` 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 initialisation, e.g. of `x(3)` above, 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:

```
double precision:: e = exp(1d0) ! OK in f2003 but not f95
```

Note: blank spaces are optional before and after `::`

14 INCLUDE

In f77 there was no standard way to include the contents of another file in your program. In f95 you may include the file `foobar.f90` with the line:

```
include 'foobar.f90' ! or include "foobar.f90"
```

Note: don't mix free and fixed source form in the same program file, but you may compile a fixed-form file (e.g. `foo.f`) and a free-form one (e.g. `bar.f90`) together, e.g. if one is the main program and the other is a collection of subroutines it uses, you might compile them with `f95 foo.f bar.f90`

15 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`.

16 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, 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))
```

In f95, to change the size of an allocated array you must `deallocate` it first. The function `allocated` tells whether the array is currently allocated, so:

```
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. The previous values of its elements will have been lost.

In f2003, the size of an allocated array changes if you assign to it a value of a different size, e.g. `x = (/666.0,x/)` increases the size of `x` by 1 and puts `666.0` in front of the previous values of `x`.

WARNING: test that last feature before relying on it. Some compilers may not have it yet.

17 New standard ("intrinsic") functions

There are many of these; I give only a few examples. They can give problems with an old f77 program, because if it has a subroutine or function whose name is the same as the name of an f95 intrinsic that the f77 standard didn't require, e.g. `sum`, then you will get the new intrinsic unless you put yours after `contains` in your main program (§18 on page 14), or in a module (§21 on page 17), or you specify yours as `external`. (Old compilers were allowed

to offer nonstandard intrinsics; I don't know if `sum` was one of them.) The same thing was needed for the same reason in the change from f66 to f77: if your f66 program had a function called `LEN`, you had to say `EXTERNAL LEN` when using an f77 compiler unless you wanted the f77 intrinsic function `LEN`.

17.1 MERGE

`Merge(truestuff,falsestuff,mask)` is one of the intrinsics not in f77, in which `truestuff` may be of any type, `falsestuff` is of the same type, and `mask` is of type logical. If any of them is an array, the others must be either arrays of the same size and shape, or scalars. 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 evaluated even though only one of them is used, so do not rewrite the `cuberoot` function in §15 on page 11 as

```
cuberoot = merge(0.0, &
     merge(t**(1.0/3.0), -((-t)**(1.0/3.0)),t>0), t=0)
```

unless you want to see if it produces an error message. Similarly, be careful with `merge` in recursive functions and subroutines (§20 on page 16) to avoid the possibility of infinite recursion.

17.2 KIND and selection of kinds

Some old programs had declarations like `real*8 x`. That was never standard f77, and it was never portable: some compilers use different numbers for a given precision. `Real(8) x` is standard f95, but it 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 often use

```
integer,parameter:: dp = kind(1d0)
real(dp) x
complex(dp) z
```

Complex double precision was not in f77, but it's always available in f95. Some f95 compilers offer even higher precision: see §17.3. If they do, they must offer it for both real and complex 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. If you ask for too many digits then `dp` will be negative, and declarations like `real(dp) x` will fail. There are also `selected_int_kind` and (in f2003) `selected_char_kind`, which is useful if there are several character sets (e.g. ISO_10646, EBCDIC, Kanji)

17.3 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.

There must be at least two real kinds. One occupies the same amount of storage as a default integer and the other (double precision) twice as much, as in f77. There may be more than two, e.g. gfortran has all four of those listed below in an x86_64 machine:

	`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
"extended precision"	18	1.2E+4932	1.1E-19	3.4E-4932
"quadruple precision"	33	1.2E+4932	1.9E-34	3.4E-4932

See http://homepages.ecs.vuw.ac.nz/~harper/fortranstuff.shtml for the programs kinds.f90 and kinds03.f90 that give these and many other results with whichever compiler you may wish to use.

17.4 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 `0`. In the same way `ubound(x)` finds upper bounds: `ubound(x)` is `(/3,9/)`, `ubound(x,2)` is `9`.

17.5 MAXVAL, MINVAL, MAXLOC and MINLOC

If `x` is a real or integer (or character in f2003) array, 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. `Minval` and `minloc` do the corresponding things for least elements.

17.6 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),'"'
 end program
```

prints this output:

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

17.7 SUM, PRODUCT, TRANSPOSE and MATMUL

If `x` is a real, integer or complex array, `sum(x)` is the sum of all its elements; `sum(x,n)` sums only over subscript number `n` and returns an array with one fewer dimension than `x`. `Product` multiplies elements instead of adding them. `Transpose(x)` transposes a two-dimensional array `x`. `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.

18 CONTAINS and explicit interfaces

Subprograms are subroutines and all functions except statement functions. *WARNING*: Fortran calls arguments of subprograms what some other languages call their parameters. Fortran's parameters are constants.

In f77 subprograms had to be defined outside your main program, usually written after its end in the same file, or in separate files. `COMMON` blocks in both the main program and the subprogram made things from one available in the other without being arguments of the subprogram. Mismatches of type (e.g. integer in one place, real in the other) or precision (e.g. double in one place but not the other) were not detected; you just got wrong answers. In f95 you may have a `contains` statement before the `end` of a main program (or of a module: see §21 on page 17), and declare subprograms between `contains` and that `end`. A "contained" subprogram such as `cuberoot` below may use anything in its "container", such as `third`, without COMMON:

```
 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
```

Such subprograms must end with `end function` or `end subroutine` and it's a good idea to put the subprogram name there too.

The compiler automatically gives such subprograms "explicit interfaces". Subprograms that don't follow `contains` normally have "implicit interfaces", but you may write "interface blocks" for them, and you sometimes have to. The books tell you when you must, and how to do it. `Contains` avoids that hassle, but raises another one: subprograms (functions or subroutines) after `contains` in a main program may not be used as arguments of subprograms. However they may if they are in a module after `contains`; see §21 on page 17.

See the books on choosing between `external` or an interface block when declaring dummy arguments that are subprograms themselves.

19 Arrays in functions/subroutines

In f77, arrays which were dummy arguments of subprograms could be given the last (*but no other*) 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 you can do this for any or all dimensions with a colon (`:`) instead of `*`, e.g.

```
real a(0:), b(:,:) ! Subscripts start at 0 for a, but 1 for b
```

Such a subprogram must have an explicit interface (see §18 on page 14). And if you want an array subscript to start somewhere other than 1 you must say so, as in `real a(0:)` above.

Arrays such as `a` and `b` are called "assumed shape". 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. If you don't, it won't usually bother because that makes programs run slower. Example:

```
integer:: i2(5)=(/ 2,4,6,8,10 /)
print '(5I4)', i2, square(i2)
contains
  function square(       n)
    integer,intent(in):: n(:)
    integer  square(size(n))
    square = n*n
  end function square
end program
```

It prints the following:

```
   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 the size of `n` was. Functions returning arrays instead of scalars didn't exist in f77; they need explicit interfaces (see §18 on page 14).

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 to swap the contents of arrays `a` and `b`, in which the array `work` is automatic:

```
 subroutine swap(a,b)
   real, dimension(:), intent(inout):: a,b
   real, dimension(size(a)):: work
   work = a
   a = b
   b = work
 end subroutine swap
```

20 Recursion

In f95 a non-elemental function or subroutine may be specified `recursive`; a recursive function needs a `result` clause. This program uses recursion to find the greatest common divisor = highest common factor of two integers. (F77 already had the function `mod`, and implied-do lists in `print` statements.)

```
implicit none
  integer m,n
  print '(3(A,I2))',(('gcd(',m,',',n,')=',gcd(m,n),&
        m=-6,6,6),n=-8,8,8)
contains
  integer recursive function gcd(a,b) result(hcf)
    integer,intent(in):: a,b
    if(b==0)then
       hcf = abs(a)
    else
       hcf = gcd(b,mod(a,b))
    end if
  end function gcd
end program
```

It prints

```
gcd(-6,-8)= 2
gcd( 0,-8)= 8
gcd( 6,-8)= 2
gcd(-6, 0)= 6
gcd( 0, 0)= 0
gcd( 6, 0)= 6
gcd(-6, 8)= 2
gcd( 0, 8)= 8
gcd( 6, 8)= 2
```

21 Modules

You may put declarations of constants, variables and subprograms between `module modname` and `end module modname`, where `modname` is the name of your module. Any subprograms must go between `contains` and `end module` just as subprograms inside a main program must go between `contains` and `end program`. The three good reasons for writing modules are:

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 (even `implicit none`). That makes available what's inside the module, including subprograms. The interfaces of subprograms in modules are automatically explicit (see §18 on page 14). Things in a module may be declared `private` and then they can be referred to from within the module but not from outside. Things may also be declared `public`. Everything is public unless otherwise specified.

To use only some of the public things (e.g. `x, y`) from a module, say

```
use modname, only: x, y
```

Many people always do that because it documents 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 can only be done with pointers.

22 Signed zero, Not a Number, and Infinity

The IEEE 754 standard distinguishes between `-0.0` and `+0.0`. F90 forbade `-0.0` to be printed with its minus sign, f95 allows it, and f2003 requires it. In an IEEE 754-compliant system `sign(a,+0.0)` gives `+abs(a)`, and `sign(a,-0.0)` gives `-abs(a)`, if `a` and `0.0` are of the same real kind.

IEEE 754 also has special real values "Not a Number", "+Infinity" which is beyond +huge, and "−Infinity" which is beyond −huge. These are usually printed as `NaN`, `+Inf` or `Inf` depending on the format, and `-Inf`, but some systems use `Infinity` instead of `Inf`.

F2003 has 5 intrinsic modules IEEE_ARITHMETIC, IEEE_EXCEPTIONS, IEEE_FEATURES, ISO_C_BINDING and ISO_FORTRAN_ENV, which you use like any other module. They give access to various IEEE and ISO features for floating-point arithmetic, error handling, and making Fortran programs interoperable with C programs.

Even if your compiler does not have those those modules it may support `NaN`, `+Inf` and `-Inf`. This little program will test for that:

```
      character(13):: cnaninf = 'NAN +INF -INF'
      integer ios
      real xnaninf(3)
      read(cnaninf,*,iostat=ios) xnaninf
      print *, merge('NaN,Inf OK','No NaN,Inf',ios==0)
      end
```

Note: this program reads with format `*` from a character variable `cnaninf`. Allowing that is new in f95; see §23.2 on page 20. Also, `cnaninf` must be a variable because the Fortran standard forbids reading from a character constant. Some compilers may let you do it but it's not a good idea to depend on nonstandard behaviour of compilers. Also the NaN you get by

reading it may be different from what illegal calculations such as 0.0/0.0 give.

WARNING: many compilers have no equivalent of Infinity for integers. 32-bit machines often have huge(1) = 2**31-1 = 2147483647, and every 32-bit pattern represents an integer n: -2147483648 $\leq$ n $\leq$ +2147483647. Often `huge(1)+1` is evaluated as `-huge(1)-1` with no error message.

That may cause problems with factorials as 13! = 6,227,020,800, but for many compilers the largest integer is 2,147,483,647. However f2008 offers the real functions `gamma(x) = (x-1)!` and `log_gamma(x) = log(abs(gamma(x)))`, which may help people needing large factorials.

23 New format features

23.1 Formats B, O, Z, I0, F0.n, ES, EN

If w is an integer representing the width, Bw, Ow or Zw formats may now be used to write integer values in binary, octal or hexadecimal form, but beware. In some systems the first bit is the sign bit, so `z'80000001'` may mean `-1`, but in other systems it means `-huge(1) = 1-2**31 = -2147483647`. As with I format, Z8 omits any leading zeros and Z8.8 puts them in.

B, O, Z, and now also I and F permit width 0 in output, to write the value in the smallest amount of space that does not give a row of asterisks. For example

```
print '(A,F0.3,A)', '"',-0.1,'"'
```

should print `"-.100"`, but some compilers give `"-0.100"` instead, because the f90 standard was poorly worded on that point. The standard committee issued an interpretation to clarify the matter, and f95 and later standards were worded in accordance with it. F0.0 format is allowed, and if there would be no digits at all then 0 must be put in before the decimal point, so `print '(A,F0.0,A)', '"',-0.1,'"'` should print `"-0."`

Beware: F0 format asks for a **lot** of output with a large number. If your compiler offers a real type with range at least 2000, try this program:

```
integer,parameter:: big=selected_real_kind(r=2000)
print '(F0.0)', huge(1.0_big)
end
```

It may print a number with 4933 digits, but at least one compiler allowing such a real kind can't. Instead, you get a run-time segmentation fault.

ES (E scientific) format is like E format except that there is one digit before the decimal point; EN (E engineering) uses an exponent that is a multiple of 3, and 1, 2 or 3 digits before the decimal point. Example:

```
real:: x(3)=1/7.0
print '(E10.1,ES10.1,EN10.1)',x
end
```

prints `0.1E+00  1.4E-01 142.9E-03`

As in f77, `Ew.dEe` with `w` and `e` $> \mathbf{0}$ and `d` $\geq \mathbf{0}$ are allowed; `e` says how many digits are used in the exponent part. In f95 both `ES` and `EN` allow `Ee`. In f2018, `w`, `e` or both may be zero, meaning that the smallest amount of space not giving a row of asterisks is used.

23.2 Format * input and output

Many of the examples in these notes use `*` instead of an explicit format. This is officially called list-directed formatting. As in f77 that means you get the compiler's choice of format. For example g95 prints `n= 666` but gfortran prints `n=  666` with this program:

```
  print *, 'n=',666
end program
```

The only new feature here is that in f95 but not in f77 you may use format * with an "internal file", i.e. a scalar or array variable of type character. See §22 on page 18 for an example, in which `cnaninf` is the internal file.

24 Derived types and defined operators

In f95 you can declare "derived types" (preferably in a module) and use them. They are particularly useful if several variables are linked and you want to operate on the whole entity. The components of a derived type need not all be of the same type themselves. One could imagine a law firm using a type `client` defined as

```
type client
  character(100) client_name, lawyer_name, client_address*200
  real(kind(1.0d0)) amount_owing
end type client
```

Note: names and amounts of money are tricky. Apparently a British bank almost sent something to "The Right Honourable the " because it hadn't allowed for names over 25 characters long. Banks must also allow for the largest possible balances, and prevent millions of fractions of cents (e.g. from interest payments) being diverted into inappropriate accounts.

A mathematical example: below is a module that defines a 3-dimensional vector as a type `v` and an operator `.x.` to find the vector product of two vectors, and a little program that uses it. Note how `%` allows you to manipulate components of the array `varay` which is itself a component of `v`, and `v(array)` turns a 3-component real array into a vector of type `v`.

```
module vectors
  implicit none
  type v
     real aray(3)
  end type v
  interface operator (.x.)
     module procedure vxv
  end interface operator (.x.)
  interface operator (+)
     module procedure vplusv
  end interface operator (+)
contains
  pure type(v) function vxv(x,y)
    type(v),intent(in) ::   x,y
    vxv = v((/x%aray(2)*y%aray(3)-x%aray(3)*y%aray(2),&
              x%aray(3)*y%aray(1)-x%aray(1)*y%aray(3),&
              x%aray(1)*y%aray(2)-x%aray(2)*y%aray(1)/))
  end function vxv

  pure type(v) function vplusv(x,y)
    type(v),intent(IN) ::      x,y
    vplusv = v(x%aray + y%aray)
  end function vplusv
end module vectors

program testv
  use vectors
  implicit none
  type(v):: v1 = v((/1,2,3/)), v2 = v((/4,5,6/))
 print "(A,T14,3F6.1)", ' v1', v1, ' v2 = v3 =', v2, &
     ' v1.x.v2', v1.x.v2, &
     ' v1.x.v2+v3', v1.x.v2+v3, &
     '(v1.x.v2)+v3',(v1.x.v2)+v3
end program testv
```

That program produced the following output:

```
 v1             1.0   2.0   3.0
 v2             4.0   5.0   6.0
 v1.x.v2       -3.0   6.0  -3.0
 v1.x.v2+v2    -6.0  12.0  -6.0
(v1.x.v2)+v2    1.0  11.0   3.0
```

Note that `v1.x.v2+v2` means `v1.x.(v2+v2)`, not, as you might have expected, `(v1.x.v2)+v2`. That is because if you "overload" an existing operator (here `+`) it keeps its original precedence, but new binary operators have the lowest possible precedence, so `.x.` cannot have the same precedence as ordinary multiplication (`*`). If you avoid that problem by defining `*` acting on two "vectors" of type `v` to do what `.x.` does, then what will you do about the scalar product of two vectors? Using `/` for it would keep the expected precedence but might confuse your readers.

25 Minimal changes from f77 to f95

Here are the minimal changes needed in an old f77 program to get the f95 bug-hunting improvements if you used neither `EXTERNAL` statements nor `BLOCK DATA` nor Hollerith strings such as `5Hstuff` .

1. If you had a `PROGRAM` statement, change it to `subroutine f77_main` .
If not, put that line before the first line of your old main program.
2. Now put these 4 lines before the `subroutine f77_main` line:

```
program f95_prog
   implicit none
   call f77-main
contains
```

3. Insert all your f77 code after `contains` .
4. Change all the `end` statements to `end subroutine` or `end function` as required.
5. Add an extra line at the end saying

```
end program f95_prog
```

If you have `BLOCK DATA` program units put them after that extra line. If you have `EXTERNAL` statements identifying block data program units, leave them where they now are. Get rid of other `EXTERNAL` statements unless they indicate subprogram (function or subroutine) names used as dummy arguments of subprograms, or a subprogram of your own that happens to have the same name as an intrinsic one (see §17 on page 11). In either case don't put the actual argument subprograms in `program f95_prog` but leave them outside it. A more complicated way, but allowing better bug-checking, is to use interface blocks (see §18 on page 14) instead of `EXTERNAL` in the subprograms that have subprograms as dummy arguments.

F77 but not f95 FORMAT statements allow Hollerith strings like `5Hstuff` which your compiler may or may not still allow. F77 already let you use `'stuff'` instead; f95 allows either that or `"stuff"`.

On changing to free source form, see §4 on page 5.