

Fortran 95 for Fortran 77 users

J.F. Harper

School of Mathematics, Statistics and Computer Science
Victoria University of Wellington.

2 May 2006

1 Introduction

Fortran has for almost 50 years been a computer language used mainly by engineers and scientists (but by few computer scientists), mainly for numerical work. Five versions were standardised and are commonly referred to as f66, f77, f90, f95 and f2003 to indicate the year. F77 superseded f66, f95 has superseded f90, and no f2003 compilers exist yet, so these notes concentrate on f77 and f95.

These notes are written to show f77 users a number of the f95 features that I found so useful that I gave up f77 except when writing a program for someone with no f95 compiler. You may use as many or as few of the new features as you wish to: almost all of f77 is also valid f95.

Some new features make programming easier, some allow the machine to detect bugs that f77 compilers cannot, and some make programs easier to read, often by reducing the number of statement labels. For example,

```
666 CONTINUE
```

forces the reader to look for `WRITE(...,END=666)`, `GO TO 666`, and various other ways in which control might have been sent to statement 666.

These notes merely scratch the surface of f95: after all they occupy only 14 pages, while the books listed below all have several hundred pages. They give many additional features of what I have described, and they explain many features of the language that I have omitted. SMSCS appears in a few places below; it means the School of Mathematics, Statistics and Computer Science.

2 Recommended books

Michael Metcalf, *Effective Fortran 77*, Oxford (1985)

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

Michael Metcalf, John Reid and Michael Cohen, *Fortran 95/2003 Explained*, Oxford (2004)

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

The first three are as concise as the subjects permit, and are comprehensive.

Adams et al. (1997) is also comprehensive, less concise, but on some matters is better as a reference book. Both it and Metcalf et al. (2004) disagree with the f95 standard in a few places, though. Google may help you find free copies of that, its three corrigenda, and the f2003 standard on the Web.

Most Fortran programs contain floating-point arithmetic. That is convenient but it has traps for young players. I recommend googling “goldberg floating point”.

There is a newsgroup called comp.lang.fortran; it deals with many questions on the language; its signal-to-noise ratio is higher than most, and some real experts frequently contribute, e.g. members of the Fortran standard-editing committee and of major computer companies’ Fortran teams, and one of the authors mentioned above.

2.1 Recommended compilers and their bugs

The first good free compiler, available on many kinds of machine, was g77, for f77 as its name suggests. There are two free f95 compilers: g95 and gfortran. I prefer g95. The University’s ITS machine mahoe also has 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 do recommend trying another 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. (I have found more bugs in g95 than in anything else, but most of them get fixed very fast when reported. At the time of writing I know of only one unfixed g95 bug, but of three unfixed bugs in another compiler.)

3 Upper and lower case

Strict f77 allowed only 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 both are always allowed. Upper and lower are 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, END in upper case, and everything else in lower case; others use lower case for everything.

4 Quotation marks

In f77 character strings had to be delimited with apostrophes (') and an apostrophe inside one had to be two together (' '). In f95 you may delimit a string with either apostrophes or quotation marks ("), so these 4 statements all print the same thing:

```
PRINT '(A)', ' Student's t test'
PRINT "(A)", ' Student's t test'
PRINT '(A)', " Student's t test"
PRINT "(A)", " Student's t test"
```

The f66 method using 17H STUDENT'S T TEST was already obsolete in f77.

5 Underscores in names

In f95 the underscore (`_`) may be used as well as letters or digits in the 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 between statements

In f95 a semicolon (`;`) ends a statement, so you may put several on a line, e.g.

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

You can then see more of your program in one computer screen but your readers may not notice what you did. Many people avoid semicolons for that reason.

7 Free source form

Fortran 77 used *fixed source form* in which various things had to be in particular places: comments marked by `*` or `C` in space 1, statement numbers in spaces 1–5, continuation from the *previous* line marked in space 6, other code in spaces 7–72. F95 still allows that, but it also offers *free source form*, in which `&` at the end of a line shows that the statement will continue on the *next* line, instead of something being put in space 6 on the next line itself. Comments are also different: see Section 8 below. Statement numbers (if used) must be the first nonblank thing on a line, other code may go on to space 132, and keywords, variable names and some other things must be separated from others by at least one blank space. So, if the `&` in line 5 of this program is in space 73, and the `+` is in space 6 on the next line:

```
INTEGER j,k,jk
j = 1
k = 2
jk = 8
PRINT *, j                                &
+ k
END
```

then it's valid f95 in either source form, but it will print 8 if compiled in fixed source form, 3 in 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 to change that.

8 Comments using !

An exclamation mark (`!`) begins an f95 comment in either free or fixed source form, anywhere except space 6 on a fixed source form line (where it would still mark

statement continuation), i.e. the rest of that line is comment for human readers, and the compiler will ignore it. Example, valid in both fixed and free source form:

```
x = sqrt(y) ! but what if y < 0 ??
```

In the rest of these notes ! will be used instead of the old C or *, which are still available in space 1 on a line but only in fixed source form, as in standard f77.

9 Relational operators: .LT. etc.

In f77, operators between numbers that return logical values (.TRUE. or .FALSE.) are all two letters between . and . They usually occur after IF, e.g. IF(x.LT.y)STOP In f95, you may still use them, but there are also symbols closer to what mathematicians use: < for .LT. > for .GT. <= for .LE. >= for .GE. /= for .NE. and == for .EQ. However .NOT. .AND. .OR. .EQV. .NEQV. are still as in f77.

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. That catches many bugs, e.g. in fixed source form

```
DO 666 i=1.9
```

gives a variable D0666I the value 1.9, but IMPLICIT NONE will reveal that D0666I wasn't declared. The real error was probably writing 1.9 instead of 1,9 but at least your attention will have been drawn to the right place in the program.

11 Loop features

11.1 DO and END DO

Instead of

```
DO 666 ...
```

```
...
```

```
666 CONTINUE
```

you may now write

```
DO ...
```

```
...
```

```
END DO
```

11.2 Integer-only index

If X is a real (floating-point) variable, then f77 lets you start a DO loop with

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

but doing that is bad numerically. Floating-point errors may build up, and the last value of x may even be over 1.0. When I tried it with g77 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. That 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.

Whenever *stuff2* is executed, if $j > 20$ then the loop ends immediately.

In f95 a loop may have no “index” variable to be incremented, but don’t let it go on forever! Example, in which *stuff1* may well change the value of x :

```
DO
  stuff1
  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, of some numerical type (integer, real, complex):

```
y = x+2          ! add 2 to each x-element
y = x*x          ! square each x-element
x(3:4) = x(1:2)! i.e. x(3)=x(1);x(4)=x(2)
```

11.5 Elemental functions

Suppose you 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 elemental functions, 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 $n \times n$ 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 statements for declaring something, telling the machine about its properties, and initialising it, e.g.

```

REAL PI
PARAMETER (PI = 3.1415927)

```

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

```
real,parameter::pi = 3.1415927
```

As in f77, the word `parameter` says that `pi` is a constant, and the program cannot change it. You may initialise variables as well as constants 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
```

A whole one-dimensional array may be given a value with `(/ /)` around the list of elements in f95; `[]` is also allowed in f2003. If it’s not a parameter, you may do that anywhere in the program, not only in a declaration. The list inside `(/ /)`

may contain an “implied-do” like the sort that already existed in DATA, READ, PRINT and WRITE statements in f77, e.g. if *y* is an array with 10 elements you may write

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

That gives *y* the values 1.0, 4.0, 9.0, ..., 100.0.

The catch about initialising when declaring is that that is done at compile time not run 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
```

13 INCLUDE

In f77 there was no standard way to include material from another file in your program. Most compilers allowed you to, but in various different ways. 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 specifying the dummy arguments of a subroutine or function, you may specify INTENT(IN) for any that are input only, INTENT(OUT) for any that are output only, or INTENT(INOUT) for those which are both. This helps the human reader sort out what is supposed to happen, and it allows the compiler to check if you got it wrong. Example:

```
REAL FUNCTION cuberoot(t)
  REAL, INTENT(IN):: t
  cuberoot = sign(1.0,t)*abs(t)**(1./3.)
END FUNCTION cuberoot
```

This 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 (but not a subprogram) must have fixed bounds, e.g. in

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

n must be a constant. 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))
```

To change the size of an allocated array, deallocate it first. There is a standard function allocated to tell whether it has been allocated, so you may write

```
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

16.1 MERGE

Among the many elemental functions (see Section 11.5 above) new in f95 is

```
merge(Tstuff, Fstuff, mask)
```

in which `Tstuff` may be of any type, `Fstuff` is of the same type, and `mask` is of type logical. If `Tstuff` is an array, `Fstuff` and `mask` must be arrays of the same size and shape. If `Tstuff` is of type character, `Fstuff` must be the same length. The function returns `Tstuff` if `mask` is true, `Fstuff` otherwise. Example:

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

sets `delta` to 1 if `i, j` are equal, 0 otherwise.

Warning: Calculating both `Tstuff` and `Fstuff` may be done or attempted even though one of them is not used, so don’t rewrite the cuberoot in Section 14 as

```
cuberoot = merge(0.0, &
merge(t**third, -((-t)**third), t>0), t=0) ! bad if t /= 0
```

because f95 (and f77 for that matter) won’t allow `(negative real)**real`.

16.2 SELECTED_REAL_KIND, KIND and CMPLX

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

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

```
REAL(dp) x
```

```
COMPLEX(dp) z
```

gives `x` and `z` the lowest precision that has at least 14 decimal significant digits: real and complex double precision in most compilers. If you ask for too many digits then `dp < 0` and the declarations of `x` and `z` will fail. (Complex double precision was not in the f77 standard, but it’s always available in f95. Some f95 compilers also provide even higher precision. If they do, they must provide it for both complex and real numbers.

In f95 and f77, `1d0` is a double precision number. In f95, `kind(y)` gives the “kind type” of any `y`, so `x` and `z` above could be declared as double precision by

```
REAL (kind(1d0)) x
```

```
COMPLEX(kind( x )) z
```

and `z` evaluated in double precision by, e.g.,

```
z = sqrt((-2.0_kind(x), 0)) ! (0,1)*1.414213562373095...
```

DCMPLX was not in standard f77 and is not in f95. Cmplx may now have 3 arguments, and `Cmplx(x,y,kind(1d0))` gives a complex double precision result for integer or real (single or double precision) `x,y`. Beware: `Cmplx(x,y)` is single precision in f95 even if `x,y` are both double precision, because that's what it was in f77.

16.3 HUGE, TINY, EPSILON and PRECISION

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

If `x` is of any real kind, `tiny(x)` is the smallest available positive number.

If `x` is of any real kind, `epsilon(x)` is the smallest number such that `1.0_kind(x) + epsilon(x) > 1.0_kind(x)`

If `x` is of any real or complex kind, `precision(x)` is the number of decimal significant figures you can expect for `x`.

All these make it easier to check how good your numerical analysis ought to be. 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
6	3.4E+38	1.2E-7	1.2E-38
15	1.8E+308	2.2E-16	2.2E-308
33	1.2E+4932	1.9E-34	3.4E-4932

(The precision values are exact, the others are correct to two significant figures.)

16.4 SIZE, LBOUND and UBOUND

If `x` is an array, `size(x)` is the 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`; `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.

16.5 MAXVAL, MINVAL, MAXLOC and MINLOC

If `x` is a real or integer array, then `maxval(x)` is its largest element, and `maxloc(x)` is the array of subscripts of that largest element. If there are two equal largest elements `maxloc` picks the first in array-element order. `minval` and `minloc` do the corresponding things for least elements. Beware: `maxloc` and `minloc` both behave as if the lower subscript bound of `x` is 1, so this program prints 2 not 1:

```
REAL:: a(0:2) = (/3,-6,1/)
PRINT*,minloc(a)
END
```

You could fix that with `PRINT*,minloc(a)+lbound(a,1)-1`

16.6 LEN, TRIM and LEN_TRIM

If x is of type character, $\text{len}(x)$ is its length (as in f77), $\text{trim}(x)$ is its value with all trailing blanks removed, and $\text{len_trim}(x)$ is $\text{len}(\text{trim}(x))$, so if you had said

```
CHARACTER x*8
```

```
x = 'foobar' ! both f77 and f95 insert blanks on the right
```

then

```
len(x) is 8,      x is 'foobar ',
len_trim(x) is 6, trim(x) is 'foobar'.
```

16.7 SUM, PRODUCT and MATMUL

If x is a real, integer or complex array, $\text{sum}(x)$ is the sum of all its elements. There is also a $\text{sum}(x, n)$ 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. $\text{Matmul}(a, b)$ calculates the matrix product if one of a, b is two-dimensional, the other is either one- or two-dimensional, and the product exists.

17 CONTAINS and explicit interfaces

Subroutines and functions (but not statement functions) are declared in subprograms. In f77 all subprograms were quite separate from your main program; they were either written after its end in the same file, or in separate files. One needed 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, and often BLOCK DATA to initialise them. Mismatches of type (e.g. integer in one place, real in the other) were not detected by the machine: you just got wrong answers.

In f95 you put a CONTAINS statement at the end of a main program (or a module: see Section 20 below), and declare subprograms between CONTAINS and END. If you do, each subprogram must end with END FUNCTION or END SUBROUTINE and it's a good idea to put the subprogram name there too. "Contained" subprograms have access to what's in the container, e.g. third inside FUNCTION cuberoot here:

```
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 automatically gives contained subprograms what are known as explicit interfaces. Other subprograms normally have implicit interfaces, but you may write explicit ones in “interface blocks” for them, and if you want to do various things allowed in f95 but not in f77 you may have to. Metcalf et al. (2004) or Adams et al. (1997) tell you when, and how to. CONTAINS avoids the hassle.

18 Arrays in functions or 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(:, :)
```

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 more slowly, but there will be an option to enforce checking. Failure to do so is a fertile breeding-ground for bugs.

In this example, *square(n)* is a function returning an array the same size as its input array *n*; you couldn’t use *n(*)* instead of *n(:)* when declaring it as a dummy argument of the function 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, but CONTAINS sees to that here.)

```
IMPLICIT NONE
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
```

That prints

```
2  4  6  8 10
4 16 36 64 100
```

19 RECURSION

In f95 a function or subroutine may call itself if you specify `RECURSIVE` when declaring it and give a function a `RESULT` clause. Factorials offer a nice illustration, although there are better ways to calculate them:

```
print*, 'fac(n) is n! for n>=0, -1 if n<0'  
! Note: the ! above isn't a comment!  
print"(2(A,I3))", &  
      ('i=',i,' fac(i)=',fac(i),i=-1,4)  
contains  
  recursive function fac(n) result(answer)  
    integer                                answer  
    integer,intent(in):: n  
    if (n<0) then  
      answer = -1  
    else if (n==0) then  
      answer = 1  
    else  
      answer = fac(n-1) * n  
    end if  
  end function fac  
end
```

That prints

```
fac(n) is n! for n>=0, -1 if n<0  
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 cannot be elemental (see Section 11.5).

20 MODULE

You may declare constants, variables and subprograms between `MODULE modname` and `END MODULE modname`. That declares a “module” called `modname`. Any subprograms 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. As with `INCLUDE`, one may be able to avoid “reinventing the wheel”.
2. It’s not easy for humans to write correct explicit interfaces (Section 17), but with modules the compiler does it for you.
3. Most `COMMON` and `BLOCK DATA` statements become superfluous, and compilers can find bugs with modules that they would miss with `COMMON` or `BLOCK DATA`.

To use a module, write

```
USE modname
```

at the beginning (after any PROGRAM, FUNCTION, SUBROUTINE, BLOCK DATA or MODULE statement but before anything else). That makes available what's inside the module, including subprograms, the interfaces of which are automatically explicit (see Section 17). There is an exception: things in a module may be labelled PRIVATE and then they can be referred to from within the module but not outside.

To use only some of the public (non-PRIVATE) things (say *x*, *y*) from a module, say

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

Many people recommend always doing that because it documents in the using program what was actually used.

You may change the names in your program of things in the module. For example, if your program has a variable *y*, and you also want to use the *y* in the module, you could rename the module's *y* as (say) *y*_{mod} in your program with

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

if you also want an ONLY clause, or

```
USE modname, ymod => y
```

if you don't.

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

21 Downloading g95 to a Unix machine

I use g95 on a Sun Sparc Solaris machine, starting from scratch with steps 1–3 below, which should have to be done only once. From time to time I download a new version by following steps 4–7. If you use Unix on something else that g95 has been ported to, you'll have to modify the `g95install` script below to suit. I am not the right person to ask about installing anything on other systems that also have g95 versions (e.g. Linux, Windows) as I don't use them. My 7 steps are:

1. In your `.cshrc` file, insert these 2 lines, of which the first gets the g95 command back even if, as sometimes happens, it has been taken off `/tmp` :

```
ln -sf ~/G95/g95-install /tmp/g95
```

```
alias g95 "env LD_LIBRARY_PATH=/usr/pkg/lib ~/bin/g95"
```

People outside SMSCS may well need a different alias command; their systems programmers should be able to advise.

2. In any convenient directory create a csh shell script called `g95install` that reads as follows for me. It will install the g95 compiler and its manual in your directory `G95`, which it creates if it has to. It puts the g95 command in the `/tmp` directory so you don't need "super-user" privileges to install it. If your system is not an SMSCS Sparc Solaris you may have to change only the two lines with # platform-dependent # on them:

```

#!/bin/csh
# csh shell script to install g95 if its tarball has
# already been downloaded to the user's home directory
set g95type = 'solaris'           # platform-dependent #
set tarball = 'g95-sparc-solaris.tgz' # platform-dependent #
cd
if !( -r $tarball ) then
    echo "You must first get" $tarball "from the g95 web site"
    exit
endif
# See if user is on a suitable machine ...
if !( $OSTYPE == $g95type ) then
    echo "Your g95 version is for a" $g95type "machine, so"
    echo "you must log in on one before entering ./g95install"
    exit
endif
# Ensure there's a directory G95 for the g95 compiler to go into
if !( -d G95 ) then
    mkdir G95
endif
# Now unpack and install g95
mv $tarball G95
cd G95
tar -zxvf $tarball
rm /tmp/g95
ln -s ~/G95/g95-install /tmp/g95
rehash

```

3. Make g95install executable with `chmod u+x g95install`
4. Find <http://www.g95.org/> on the Web
5. Click on Download prebuilt binaries
6. Click on the HTTP entry for your machine and download the tgz file.
7. Enter at the Unix command line `./g95install`
8. Assuming that that all worked, enter `g95 program-file` where *program-file* is the name of the file containing a Fortran (77 or 95) program. If it compiled OK, run it (with the command `./a.out` unless you used the `-o` option).