Data Types

Simple Data Types

`integer(`*specs*`)[,`*attrs*`] :: i`	integer
`real(`*specs*`)[,`*attrs*`] :: r`	real number
`complex(`*specs*`)[,`*attrs*`] :: z`	complex number
`logical(`*specs*`)[,`*attrs*`] :: b`	boolean variable
`character(`*specs*`)[,`*attrs*`] :: s`	string
`real, parameter :: c = 2.9e1`	constant declaration
`real(idp) :: d; d = 1.0d0`	double precision real
`s2=s(2:5); s2=s(:5); s2=s(5:)`	substring extraction

attributes: `parameter, pointer, target, allocatable, dimension, public, private, intent, optional, save, external, intrinsic`

specs: `kind=...`, **for character:** `len=...`

double precision: `integer, parameter :: idp = kind(1.0d0)`

Derived Data Types

`type person_t`	define derived data type
`  character(len=10) :: name`	
`  integer :: age`	
`end type person_t`	
`type group_t`	
`  type(person_t),allocatable &`	*F2008*: allocatable ...
`  & :: members(:)`	...components
`end type group_t`	
`name = group%members(1)%name`	access structure component

Arrays and Matrices

`real :: v(5)`	explicit array, index 1..5
`real :: a(-1:1,3)`	2D array, index -1..1, 1..3
`real, allocatable :: a(:)`	"deferred shape" array
`a=(/1.2,b(2:6,:),3.5/)`	array constructor
`v = 1/v + a(1:5,5)`	array expression
`allocate(a(5),b(2:4),stat=e)`	array allocation
`dealloate(a,b)`	array de-allocation

Pointers (avoid!)

`real, pointer :: p`	declare pointer
`real, pointer :: a(:)`	"deferred shape" array
`real, target :: r`	define target
`p => r`	set pointer p to r
`associated(p, [target])`	pointer assoc. with target?
`nullify(p)`	associate pointer with NUL

Operators

`.lt. .le. .eq. .ne. .gt. .ge.`	relational operators
`<  <=  ==  /=  >  >=`	relational op aliases
`.not. .and. .or. .eqv. .neqv.`	logical operators
`x**(-y)`	exponentiation
`'AB'//'CD'`	string concatenation

Control Constructs

if `(...)` *action*	if statement
if `(...) then`	if-construct
`  `*block*	
`else if (...) then; `*block*	
`else; `*block*	
`end if`	
select case `(number)`	select-construct
`  case (:0)`	everything up to 0 (incl.)
`    `*block*	
`  case (1:2); `*block*	number is 1 or 2
`  case (3); `*block*	number is 3
`  case (4:); `*block*	everything up from 4 (incl.)
`  case default; `*block*	fall-through case
`end select`	
`outer:` **do**	controlled do-loop
`  inner: do i=from,to,step`	counter do-loop
`    if (...) cycle inner`	next iteration
`    if (...) exit outer`	exit from named loop
`  end do inner`	
`end do outer`	
do while `(...);`*block*`;end do`	do-while loop

Program Structure

program `myprog`	main program
`  use foo, lname => usename`	used module, with rename
`  use foo2, only: `*[only-list]*	selective use
`  implicit none`	require variable declaration
`  interface;...;end interface`	explicit interfaces
`  `*specification-statements*	var/type declarations etc.
`  `*exec-statements*	statements
`  stop 'message'`	terminate program
`contains`	
`  `*internal-subprograms*	subroutines, functions
`end program myprog`	
module `foo`	module
`  use bar`	used module
`  public :: f1, f2, ...`	list public subroutines
`  private`	make private by default
`  interface;...;end interface`	explicit interfaces
`  `*specification statements*	var/type declarations, etc.
`contains`	
`  `*internal-subprograms*	"module subprograms"
`end module foo`	
function `f(a,g) result r`	function definition
`  real, intent(in) :: a`	input parameter
`  real :: r`	return type
`  interface`	explicit interface block
`    real function g(x)`	dummy var g is function
`      real, intent(in) :: x`	
`    end function g`	
`  end interface`	
`  r = g(a)`	function call
`end function f`	
`recursive function f(x) ...`	allow recursion
`elemental function f(x) ...`	work on args of any rank

subroutine `s(n,i,j,a,b,c,d,r,e)`	subroutine definition
`  integer, intent(in) :: n`	read-only dummy variable
`  integer, intent(inout) :: i`	read-write dummy variable
`  integer, intent(out) :: j`	write-only dummy variable
`  real(idp) :: a(n)`	explicit shape dummy array
`  real(idp) :: b(2:,:)`	assumed shape dummy array
`  real(idp) :: c(10,*)`	assumed size dummy array
`  real, allocatable :: d(:)`	deferred shape (*F2008*)
`  character(len=*) :: r`	assumed length string
`  integer, optional :: e`	optional dummy variable
`  integer :: m = 1`	same as `integer,save::m=1`
`  if (present(e)) ...`	presence check
`  return`	forced exit
`end subroutine s`	
`call s(1,i,j,a,b,c,d,e=1,r="s")`	subroutine call

Notes:

- explicit shape allows for reshaping trick (no copies!): you can pass array of any dim/shape, but matching size.
- assumed shape ignores lbounds/ubounds of actual argument
- deferred shape keeps lbounds/ubounds of actual argument
- subroutines/functions may be declared as `pure` (no side effects)

Use of interfaces:

- *explicit interface* for external or dummy procedures

`interface`	
`  `*interface body*	sub/function specs
`end interface`	

- *generic/operator/conversion interface*

`interface `*generic-spec*	
`  module procedure `*list*	internal subs/functions
`end interface`	

generic-spec can be any of the following:

1. "generic name", for overloading routines
2. operator name (`+` `-`, etc) for defining ops on derived types
   You can also define new operators names, e.g. `.cross.`
   Procedures must be one- or two-argument functions.
3. `assignment (=)` for defining assignments for derived types.
   Procedures must be two-argument subroutines.

The *generic-spec* interfaces should be used inside of a module; otherwise, use full sub/function specs instead of module procedure list.

Intrinsic Procedures

Transfer and Conversion Functions

`abs(a)`	absolute value
`aimag(z)`	imag. part of complex z
`aint(x, kind), anint(x, kind)`	to whole number real
`dble(a)`	to double precision
`cmplx(x, y, kind)`	create `x + i y`
`cmplx(x, kind=idp)`	real to dp complex
`int(a, kind), nint(a, kind)`	to int (truncated/rounded)
`real(x, kind)`	to real (i.e. real part)
`char(i, kind), achar(i)`	char of ASCII code
`ichar(c), iachar(c)`	ASCII code of character
`logical(l, kind)`	change kind of logical `l`
`ibits(i, pos, len)`	extract sequence of bits
`transfer(source, mold, size)`	reinterpret data

Arrays and Matrices

`allocated(a)`	check if array is allocated
`lbound(a,dim)`	lowest index in array
`ubound(a,dim)`	highest index in array
`shape(a)`	shape (dimensions) of array
`size(array,dim)`	extent of array along dim
`all(mask,dim)`	all `.true.` in logical array?
`any(mask,dim)`	any `.true.` in logical array?
`count(mask,dim)`	number of true elements
`maxval(a,d,m)`	max value in masked array
`minval(a,d,m)`	min value in masked array
`product(a,dim,mask)`	product along masked dim
`sum(array,dim,mask)`	sum along masked dim
`merge(tsrc,fsrc,mask)`	combine arrays as mask says
`pack(array,mask,vector)`	packs masked array into vect.
`unpack(vect,mask,field)`	unpack `vect` into masked field
`spread(source,dim,n)`	extend source array into dim.
`reshape(src,shp,pad,ord)`	make array of shape from src
`cshift(a,s,d)`	circular shift
`eoshift(a,s,b,d)`	"end-off" shift
`transpose(matrix)`	transpose a matrix
`maxloc(a,mask)`	find pos of max in array
`minloc(a,mask)`	find pos of min in array

Computation Functions

`ceiling(a), floor(a)`	to next higher/lower int
`conjg(z)`	complex conjugate
`dim(x,y)`	max(x-y, 0)
`max(a1,a2,..), min(a1,..)`	maximum/minimum
`dprod(a,b)`	dp product of sp a, b
`mod(a,p)`	a mod p
`modulo(a,p)`	modulo with sign of a/p
`sign(a,b)`	make sign of a = sign of b
`matmul(m1,m2)`	matrix multiplication
`dot_product(a,b)`	dot product of vectors

more: `sin, cos, tan, acos, asin, atan, atan2, sinh, cosh, tanh, exp, log, log10, sqrt`

Numeric Inquiry and Manipulation Functions

`kind(x)`	kind-parameter of variable `x`
`digits(x)`	significant digits in model
`bit_size(i)`	no. of bits for int in model
`epsilon(x)`	small pos. number in model
`huge(x)`	largest number in model
`minexponent(x)`	smallest exponent in model
`maxexponent(x)`	largest exponent in model
`precision(x)`	decimal precision for reals in
`radix(x)`	base of the model
`range(x)`	dec. exponent range in model
`tiny(x)`	smallest positive number
`exponent(x)`	exponent part of x in model
`fraction(x)`	fractional part of x in model
`nearest(x)`	nearest machine number
`rrspacing(x)`	reciprocal of relative spacing
`scale(x,i)`	`x b**i`
`set_exponent(x,i)`	`x b**(i-e)`
`spacing(x)`	absolute spacing of model

String Functions

`lge(s1,s2), lgt, lle, llt`	string comparison
`adjustl(s), adjustr(s)`	left- or right-justify string
`index(s,sub,from_back)`	find substr. in string (or 0)
`trim(s)`	s without trailing blanks
`len_trim(s)`	length of `trim(s)`
`scan(s,setd,from_back)`	search for any char in set
`verify(s,set,from_back)`	check for presence of set-chars
`len(string)`	length of string
`repeat(string,n)`	concat n copies of string

Bit Functions

`btest(i,pos)`	test bit of integer value
`iand(i,j),ieor(i,j),ior(i,j)`	and, xor, or of bit in 2 integers
`ibclr(i,pos),ibset(i,pos)`	set bit of integer to 0 / 1
`ishft(i,sh),ishftc(i,sh,s)`	shift bits in i
`not(i)`	bit-reverse integer

Misc Intrinsic Subroutines

`date_and_time(d,t,z,v)`	put current time in `d,t,z,v`
`mvbits(f,fpos,len,t,tpos)`	copy bits between int vars
`random_number(harvest)`	fill harvest randomly
`random_seed(size,put,get)`	restart/query random generator
`system_clock(c,cr,cm)`	get processor clock info

Input/Output

Format Statements

`fmt = "(F10.3,A,ES14.7)"`	format string
`Iw` `Iw.m`	integer form
`Bw.m` `Ow.m` `Zw.m`	binary, octal, hex integer form
`Fw.d`	decimal form real format
`Ew.d`	exponential form (`0.12E-11`)
`Ew.dEe`	specified exponent length
`ESw.d` `ESw.dEe`	scientific form (`1.2E-10`)
`ENw.d` `ENw.dEe`	engineer. form (`123.4E-12`)
`Gw.d`	generalized form
`Gw.dEe`	generalized exponent form
`Lw`	logical format (`T`, `F`)
`A` `Aw`	characters format
`nX`	horizontal positioning (skip)
`Tc` `TLc` `TRc`	move (absolute, left, right)
`r/`	vert. positioning (skip lines)
`r(...)`	grouping / repetition
`:`	format scanning control
`S` `SP` `SS`	sign control
`BN` `BZ`	blank control (blanks as zeros)

w full length, *m* minimum digits, *d* dec. places, *e* exponent length, *n* positions to skip, *c* positions to move, *r* repetitions

Argument Processing / OS Interaction

```
n = command_argument_count()
call get_command_argument(2, value) ! get 2nd arg
call get_environment_variable(name,            &
&    value, length, status, trim_name) ! optional
call execute_command_line(command,             &
&    wait, exitstat, cmdstat, cmdmsg)  ! optional
```

These are part of *F2003/F2008*. Older Fortran compilers might have vendor extensions: `iargc`, `getarg`, `getenv`, `system`

Reading and Writing to Files

`print '(I10)', 2`	print to stdout with format
`print *, "Hello World"`	list-directed I/O (stdout)
`write(*,*) "Hello World"`	list-directed I/O (stdout)
`write(unit, fmt, spec) list`	write list to unit
`read(unit, fmt, spec) list`	read list from unit
`open(unit, specifiers)`	open file
`close(unit, specifiers)`	close file
`inquire(unit, spec)`	inquiry by unit
`inquire(file=filename, spec)`	inquiry by filename
`inquire(iolength=iol) outlist`	inquiry by output item list
`backspace(unit, spec)`	go back one record
`endfile(unit, spec)`	write eof record
`rewind(unit, spec)`	jump to beginning of file

I/O Specifiers (`open` statement)

`iostat=error`	save int error code to `error`
`err=label`	label to jump to on error
`file='filename'`	name of file to open
`status='old' 'new' 'replace' 'scratch' 'unknown'`	status of input file
`access='sequential' 'direct'`	access method
`form='formatted' 'unformatted'`	formatted/unformatted I/O
`recl=integer`	length of record
`blank='null' 'zero'`	ignore blanks/treat as 0
`position='asis' 'rewind' 'append'`	position, if sequential I/O
`action='read' 'write' 'readwrite'`	read/write mode
`delim='quote' 'apostrophe' 'none'`	delimiter for char constants
`pad='yes' 'no'`	pad with blanks

close-specifiers: `iostat, err, status='keep' 'delete'`

inquire-specifiers: `access, action, blank, delim, direct, exist, form, formatted, iostat, name, named, nextrec, number, opened, pad, position, read, readwrite, recl, sequential, unformatted, write, iolength`

backspace-, endfile-, rewind-specifiers: `iostat, err`

Data Transfer Specifiers

`iostat=error`	save int error code to `error`
`advance='yes' 'no'`	new line?
`err=label`	label to jump to on error
`end=label`	label to jump to on EOF
`eor=label`	label for end of record
`rec=integer`	record number to read/write
`size=integer-variable`	number of characters read

For a complete reference, see:

⇒ Adams, Brainerd, Martin, Smith, Wagener, *Fortran 90 Handbook*, Intertext Publications, 1992.

There are also editions for Fortran 95, and Fortran 2003.

For Fortran 2008 features, please consult:

⇒ Reid, *The new features of Fortran 2008.* ACM Fortran Forum 27, 8 (2008).

⇒ Szymanski. Mistakes in Fortran that might surprise you: `http://t.co/SPa0Y5uB`