Fortran

(FORmula TRANslator) History

• FORTRAN vs. Fortran
• 1954 FORTRAN first successful high level language – John Backus (IBM)
• 1958 FORTRAN II (Logical IF, subroutines, functions)
• 1961 FORTRAN IV
• 1966 FORTRAN '66 - First standard language (1972)
• FORTRAN '77 (FORTRAN V) - again standard (1980).
FORTRAN 77 Added

- DO loops with a decreasing index
- Block IF statements
  IF…THEN…ELSE…ELSEIF
- Pretest of DO loops, before F77 DO loops were always executed at least once
- CHARACTER data type
- Apostrophe delimited character string
- Main program termination without a STOP statement.

FORTRAN 77 Restrictions

- Each line 72 characters long
- First 5 columns for “line” numbers
- 6th column for continuation
- Columns 73-80 “card” numbers
- Comments initiated in first column with a C
- Only upper case letters anywhere
- Variable names only 6 characters long
- No inline commenting
- Reliance on “unsafe” storage and sequence association features (COMMON blocks and EQUIVALENCE statements)
Fortran 90 (1991) Added

- Free format source code form (up to 132 columns per line)
- More than one statement per line
- Inline comments
- Upper and lower case letters
- Variable names up to 31 characters, including 
- Modern control structures (CASE, DO WHILE)
- Array notation (array sections, array operators, reduction, etc.)
- Dynamic memory allocation and deallocation
- Pointers

Fortran 90 (1991) Added

- Data attributes and structures
- Procedures and procedure interfaces (including recursion and optional arguments)
- Derived Types and Operator overloading
- Keyword argument passing, INTENT (in, out, inout)
- Numeric precision and range control (for portability)
- Modules
- Improved I/O
- No longer necessary to pass array dimension into subroutine
(FORmula TRANslator) History

- 1992 – HPF (High Performance Fortran)
- 1997 - Fortran ’95 – new standard designed to rectify ambiguities in F90 and include some desirable extensions (from HPF)
- 1996 – F, a highly modernized subset of F90
- Fortran 2000 (F2k)– working draft; scheduled for release in 2002….???

Fortran 95 from Sun Added

- Global program checking across routines for consistency of arguments, commons, parameters, etc.
- Optimized automatic and explicit loop parallelization for multiprocessor systems
- OpenMP parallelization directives
- Cray-style parallelization directives, including TASKCOMMON
- Global, peephole, and potential parallelizations for high performance applications
- Common calling conventions on Solaris to permit routines written in C or C++ to be combined with Fortran programs
- Support for 64-bit enabled Solaris environments
- Call-by-value using %VAL
- Compatibility between Fortran 77 and Fortran 95 programs and object binaries
- Interval Arithmetic programming
- Some “Fortran 2000” features, including Stream I/O
Which one?

• Code we will be working with was written in FORTRAN IV (i.e., FORTRAN ’66)
• We have compilers for
  – FORTRAN 77 (Forte 7 as a compiler option)
  – Fortran 90 (Forte 7 as a compiler option)
  – Fortran 95 (Forte 7)
• Backward compatibility, but you may use whichever you’d like and change languages if you’d like. However, it’s a large piece of code and it must produce identical output (within the ranges specified by the manual)!

Data Types

• Numerical
  – Integers (INTEGER)
  – Reals, including REAL, DOUBLE PRECISION, COMPLEX
• Character (CHARACTER) (F77+)
  – usually grouped to form a string
• Logical (LOGICAL)
  – .TRUE. or .FALSE.
Literals

- Integer 1 0 -1 42 365
- Real 1.0 0.0 -1.0 42.0 365.0 365.25
  3.14159 5.6E-10 5.6D-20
- Character 'A' 'b' 'l' '0' '-'
  - string 'ABC' '123' '-l'
    'Rochester Institute of Technology'
- Logical .TRUE. .FALSE.

Variables

- Declared at beginning of program
- Explicit typing or explicit type declaration
- Otherwise, implicit typing
  - integers beginning I, J, K, L, M, N
  - anything else is real
- Bad idea to rely on implicit typing, so use
  - IMPlicit NONE (F77+ at least)
- NOTE: Before F90, identifiers were limited to 6 letters! All caps.
Assignment

• `<variable> = <expression>`
  – days = 31
  – temperature = 4.2
  – leapyear = .TRUE.

• Automatic Mode Conversion
  – I = 14./3. (=4)
  – R = 4/3 (=1.0)

Operators

• ** raise to the power
• * multiply
• / divide
• + add
• - subtract
**Precedence**

- Parentheses
- Exponentiation (**)
- Multiplication and division, left to right
  - \(4.0 / 2.0 / 2.0\) gives 1.0
- Addition and subtraction, left to right
- Expression examples
  - \(\pi = 3.14159\)
  - \(\text{area} = \pi \times r^{2}\)
  - \(s = u \times t + 0.5 \times a \times t^{2}\)

**Typed Arithmetic**

- If all reals, applies rules of real arithmetic
- If all integers, applies rules of integer arithmetic
  - all fractional parts truncated
  - \(5 / 2\) is 2
  - \(3 / 4\) is 0
- Compiler will promote as necessary
  - \(5 / 2.0\) is 2.5
- Choice of arithmetic made at operator level
  - \(5.0 + 3 / 2\) is 6.0
- Automatic mode conversion during assignment
  (Note: will not tell you if you are losing precision)
Functions

- Takes parameters in parentheses
- Returns a single value
- Intrinsic functions (built-in)
  - sqrt, sin, cos, log, exp, abs, acos,…
- Write your own (...later...)
- Use ones from a library

Relational Operators

- .EQ. “equal to”
- .NE. “not equal to”
- .LT. “less than”
- .GT. “greater than”
- .LE. “less than or equal to”
- .GE. “greater than or equal to”
Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.EQ.</td>
<td>“equal to”</td>
</tr>
<tr>
<td>.NE.</td>
<td>“not equal to”</td>
</tr>
<tr>
<td>.LT.</td>
<td>“less than”</td>
</tr>
<tr>
<td>.GT.</td>
<td>“greater than”</td>
</tr>
<tr>
<td>.LE.</td>
<td>“less than or equal to”</td>
</tr>
<tr>
<td>.GE.</td>
<td>“greater than or equal to”</td>
</tr>
</tbody>
</table>

F90 adds:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>“equal to”</td>
</tr>
<tr>
<td>!=</td>
<td>“not equal to”</td>
</tr>
<tr>
<td>&lt;</td>
<td>“less than”</td>
</tr>
<tr>
<td>&gt;</td>
<td>“greater than”</td>
</tr>
<tr>
<td>&lt;=</td>
<td>“less than or equal to”</td>
</tr>
<tr>
<td>&gt;=</td>
<td>“greater than or equal to”</td>
</tr>
</tbody>
</table>

Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AND.</td>
<td>Logical and</td>
</tr>
<tr>
<td>.OR.</td>
<td>Logical or</td>
</tr>
<tr>
<td>.NOT.</td>
<td>Logical not</td>
</tr>
<tr>
<td>.EQV.</td>
<td>Equivalent (ex., .FALSE. .EQV. .FALSE. is .TRUE.) (F90?)</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>Not equivalent (F90?)</td>
</tr>
</tbody>
</table>
IF

• Executes single command if logical expressions evaluates as .TRUE. (Only 1 before F77)
  – Ex. IF ( A .EQ. 0.0 ) B = 1.0

• IF / THEN / ENDIF for a block of statements (F77+)
  – Ex. if ( a .eq. 0.0 ) then
    b = 1.0
    c = x + y
    endif

Else (F77+)

• if ( a .eq. 0.0 ) then
  b = c / a
else
  b = 0.0
endif
Arrays

- Has type and name, plus an index count
  - (F90+) real temperature(365)
  - (Pre F90) REAL TEMPER
  - DIMENSION TEMPER (365)
  - gives 365 reals
  - temperature(1)
  - temperature(2)
  - temperature(3)
  - ...
  - temperature(365)

Multidimensional Arrays

- eg. 1 degree grid across earth’s surface
  - real temperature(360,180)

- eg. same for fifty years of monthly data
  - real temperature(360,180,12,50)

- Stored in COLUMN MAJOR order!
DO loops

F90+
real value(50), total, mean
integer i
...
total = 0.0
do i = 1, 50
total = total + value(i)
end do
mean = total / 50.0

F66/F77
REAL VALUE(50), TOTAL, MEAN
(I is implicitly integer)
...
TOTAL = 0.0
DO 10 I = 1, 50
TOTAL = TOTAL + VALUE(I)
10 CONTINUE
MEAN = TOTAL / 50.0

Varieties of do

• do while ( logical_expression )
  body_of_do_while
end do

• do index=start, end, by)
  body_of_do
end do

• do stmt_# index=start, end, by)
  body_of_do
Stmt_# continue
WHILE loop (F77 extended+)

do while (a .gt. 0)
    b = b + a * d
    a = a - b
end do

I/O

- F77+
  - print *, mean
  - print *, 'The mean is', mean
- Before F77 and is still available
  WRITE(6, 5200) MEAN
  5200 FORMAT(11HTHE MEAN IS, 1X,
           1 F9.2)

NOTE: 6 is the Logical Unit number for the screen
Input

- F77+
  print *, ’Give me a number’
  read *, x
  print *, ’You typed’, x
  *means interactive channel

- PRE F77
  READ(5, 40) X
  400 FORMAT(I 5)

NOTE: 5 is the Logical Unit number for the keyboard

Array I/O – equivalent; notice column major order!

integer data(10,10)
.....
read *, data
.....
do j=1, 10
  read *, (data(i,j), i=1, 10)
end do
.....
read *, ((data(i,j), i=1, 10), j=1, 10)
**Formatted I/O**

- FORMAT statement must have a label
- Format images, one for each item
  - i = integer, f = floating point, e = scientific, d=double precision
    - 1000 format (i3,i5)
    - 1100 format (f8.3)
- Repeat count precedes letter
  - 5000 format (i4,5f6.2)
- Can also repeat groups
  - 3500 format (i4,5(f6.2,i3))

**Formatted I/O**

- Literal strings
  - 2200 format('The mean of ',i3, ' items was ',f7.3)
- Character strings use ‘a’ format image
  - with width, padded
  - without width, as is
- 1500 format (a20,f8.3)
- Pre F77+, H for Hollerith
- Lots of options available…many lectures just on I/O to hit them all!
read, print, write - Varieties of IO

**• Read Examples**
- READ (5, 100) X,Y
100 FORMAT (2F20.5))
- READ(9, FMT=“(2F20.5)”, IOSTAT=IEND) x, y
- READ(UNIT=5, FMT=“(2F20.5)”) 
- READ *, x, y
- READ(5, FORMAT6) X,Y (where FORMAT6 = ‘(2F20.5)’)

**• WRITE Examples**
- WRITE (6, 100) X,Y
100 FORMAT (2F20.5))
- WRITE(9, FMT=“(2F20.5)”, IOSTAT=IEND) x, y
- WRITE(UNIT=6, FMT=“(2F20.5)”) 
- WRITE(*, “(2F20.5)”) X, Y

**• PRINT Examples**
- PRINT *, x, y
- PRINT “(2F20.5)”, x, y
Spacing

• / is equivalent to \n in C
  2100 format('This is line 1', /, 'This is line 2')
• X format type for blank spaces
  2220 format(1x, 'Mean = ', f7.3)
• Column 1 character controlled vertical spacing
  (before F90)
  – Blank space down one line then print
  – 0 space down two lines then print
  – 1 advance to the first line of the next page
  – + no advances before printing (allows overprinting)

Go To

10 read(1,1000) a,b,c  
   if (a.lt.0) go to 20
   goto 10
20 continue

Can be replaced with
  (F90+)
  do
    read(1,1000) a,b,c
  until (a.lt.0)
Program Structure

- End program with END statement
- Follow with sub-programs
  - subroutines
  - functions

program myprog  (F90+)
  ...
  STOP  (<F77)
end
(subprograms here)

Subroutines

- Arbitrary number of parameters
- No return value
- but can alter value of variables passed to them, i.e., passed by reference/address!!!!

  call findextremes(365, temperature, coolest, warmest)
subroutine findextremes( n, values, low, high)
    integer n
    real values(*)
    real low,high
    integer i
    low = values(1)
    high = values(1)
    do i=1,n
        if(values(i).lt.low) low = values(i)
        if(values(i).gt.high) high = values(i)
    end do
end

real function mean(n, values)
    integer n
    real values(*)
    integer i
    real total
    total = 0.0
    do i=1,n
        total = total + values(i)
    end do
    mean = total / n
end
COMMON Blocks (<F90)

- Data shared between program units
- Variables can be renamed in each section!
- COMMON /blockname/ list of variables
- OR COMMON list of variables (blank common)
- First before any executable statements
- May be initialized through BLOCK DATA subprogram only.

Case statement

```plaintext
select case (expression)
  case (case_selector)
    block_of_statements
  case (case_selector)
    block_of_statements
  ....
  case default
    block_of_statements
end select
```
Computed Goto

The case statement replaces the computed go to statement:

\[
goto (stmt\#1, stmt\#2, \ldots, stmt\#n), int\_expression
\]

where \( int\_expression \) must evaluate to be \([1..n]\)
! This is a F90+ comment
program example
  integer i, j, k
  real    r, s, t
  i = 10  ! inline comment F90+
  j = 20
  k = 30
  r = 0.0
  s = 10.0
  t = 20.0   ! & below mean continuation!
print *, 'Initial Values, i=', i, ' j=', j, &
       ' k=', k, ' r =', r, ' s=', s, ' t=', t
print *
call sub( i, j, k )
print *, 'After call to sub,  i=', i, ' j=', j, &
       ' k=', k
print *
r = func( s, t )
print *, 'After func invoked, r= ', r, ' s=', s, &
       ' t=', t
end

Example Program

subroutine sub( m, n, o )
  integer m, n, o
  print *,
    'Entering sub,  m=', m,&
    ' n=', n, ' o=', o
  n = m
  o = m
  print *,
    'Exiting sub,  m=', m,&
    ' n=', n, ' o=', o
end

Example Program Continued

real function func( p1, p2 )
  real p1, p2
  print *
    'Entering func, ' &
    ' p1=', p1, &
    ' p2=', p2
  func = p1 + p2
  print *
    'Exiting func, func=', &
    ' p1=', p1, &
    ' p2=', p2
end
Example Program Output

Initial Values, i= 10  j= 20  k= 30  r = 0.0E+0
s= 10.0  t= 20.0

Entering sub, m= 10  n= 20  o= 30
Exiting sub,  m= 10  n= 10  o= 10
After call to sub,  i= 10  j= 10  k= 10

Entering func,  p1= 10.0  p2= 20.0
Exiting func,  func= 30.0  p1= 10.0  p2= 20.0
After func invoked,  r= 30.0  s= 10.0  t= 20.0

To compile and run

- The Fortran compiler(s) should already be available to you; type
  - which f95
- To compile
  - f77/f90/f95 (-g -free -pg -o file) file.f
  - f95 (-g -pg -o file) file.f95
  - See Makefile for “best” optimization parameters.
- To execute:
  - file <input >output
  - Default name for file is a.out
Fortran vs. C

• C better for dealing with character data or purely integer data, OS development/support

• Fortran
  – Better at I/O, numerics and parallelization
  – Allows variably-dimensioned array arguments in subroutines
  – Rich set of generic-precision intrinsic functions
  – Built-in complex arithmetic
  – Exponentiation operator
  – Better diagnostic messages from compiler
  – May pass all variables by reference – fastest
  – Cannot alias arguments when calling subroutine; each passed argument list must have distinct entries
  – Math library linkage automatic

Fortran vs. C

• F77
  – No pointers or recursion => automatic code optimization
  – Static storage allocation
  – Easier to learn

• F90+
  – array indices may start and end at an arbitrary integer
  – Array notation
  – Automatic selection of numeric data types having a specified precision and range