Fortran 90

An Overview of the New Fortran Language

Wolfgang V. Walter

Institute of Applied Mathematics

University of Karlsruhe

- History of Fortran
- Available Implementations of Fortran 90
- Overview of New Language Features
- Sample Programs and Modules
- Summary
- Fortran 90 Literature

History of Fortran

- FORTRAN 66 (ANSI/ISO 1966): first programming language standard
- FORTRAN 77 (ANSI/ISO 1978): modest revision of FORTRAN 66
- further development as "Fortran 8x": major revision necessary
- technical work: ANSI/X3J3, supervision: ISO/IEC JTC1/SC22/WG5
- many difficulties, changing world, increasing demands
- "Super-Fortran à la Ada" or "Small is beautiful"?
- First Public Review in 1987: complexity too great?
- X3J3 deadlock resolved at ISO/WG5 meeting in Paris in Sept. 1988
- Second Public Review in 1989: largely favorable
- technical content of standard finalized in 1990 $\Rightarrow$ "Fortran 90"
- Third Public Review in 1990: mostly a formality
- single world-wide Fortran standard? ANSI standard = ISO standard?
- International Standard: "Fortran 90", ISO/IEC 1539 : 1991 (Aug. 1991)
- American Standard: "Extended Fortran", ANSI X3.198-1992 identical to international standard, but FORTRAN 77 (ANSI X3.9-1978) remains in effect simultaneously (only in USA)
- European Standard: "Fortran 90", EN 21539 : 1992 and all other national standards identical to international standard (1992/93)
- next standard planned for 1996: clarifications and corrections only (?)

Available Implementations of Fortran 90

- Numerical Algorithms Group Ltd.: NAGWare – f90 Compiler
  first full implementation of Fortran 90
  generates C code for workstations and mainframes
- Salford Software Ltd./NAG Ltd.: NAGWare – FTN90 Compiler
  generates native code for IBM PCs and compatibles
  (available from h.o. Computer, Köln in Germany)
- ParaSoft Corp.: Fortran 90 Translator pf90
  translates Fortran 90 to FORTRAN 77
  available for vector processors, mainframes, workstations, PCs
- Pacific-Sierra Research Corp.: VAST-90
  translates Fortran 90 to FORTRAN 77 — and vice versa
  available for some workstations
- Edinburgh Portable Compilers Ltd.: EPC Fortran-90 Compiler
  generates native code for Intel 80x86, Sun SPARC, Motorola 68K and 88K, and IBM RS/6000
  available for certain workstations and PCs (?)
- MasPar: exclusive use of Fortran 90 for parallel programming (?)
- Convex, Cray, DEC, IBM, Lahey, SGI, Sun, and others appear committed to provide full implementations of Fortran 90 with native code generation in 1993 or 1994 (?)
- many existing FORTRAN 77 compilers offer some Fortran 90 features, in particular array and syntax extensions

Overview of Fortran 90

- Fortran 90 is a superset of FORTRAN 77
- all FORTRAN 77 programs are valid Fortran 90 programs and should compile and run as expected
- complete modernization of Fortran (subject to strict compatibility with FORTRAN 77)
- major additions, extensions and improvements in all areas
- powerful new features and concepts:
  - free source form
  - improved, block-structured syntax
  - array processing
  - assumed-shape arrays
  - allocatable arrays
  - dynamic storage
  - pointers
  - optional arguments
  - recursion
  - modules
  - derived types (user-defined data structures)
  - user-defined operators and assignment
  - explicit and generic interfaces
  - procedure, operator and assignment overloading

Lexical Enhancements

- long names with up to 31 characters
- underscore "_" allowed in names
- new syntax for relational operators: <, <=, ==, >=, >, /=
- in-line comments introduced by "!" (extend to end of line)
- multiple statements on one line separated by ";" (optional at end of line)
- " and ' may be used interchangeably to delimit character constants
- statements may be continued on up to 39 continuation lines
- free source form:
  - no special columns for comments, labels, continuation lines
  - up to 132 characters per line
  - blanks significant, lexical tokens must be contiguous
  - continuation line announced by "&" at end of preceding line
  - continuation position may be indicated by "&" on continuation line

Examples:

```
CHARACTER (LEN=60) ::  this_name_is_31_characters_long
REAL ::  x, y, z

x = 3.5 ;  y = 4.5 ;  z = 5.5 ; ! end-of-line comment
IF ( x < y .AND. z > y .OR.   & ! initial line
     x == z .AND. y /= 0.0 )  & ! 1st continuation line
  z = x + y + z                 ! 2nd continuation line

this_name_is_31_characters_long = "This sentence is &
  &more than 31 characters long, isn't it?"
```

Block-Structured Syntax

- ```
  PROGRAM  My_First_Fortran_90_Program
    WRITE (*,*) "This is my first Fortran 90 program."
  END PROGRAM  My_First_Fortran_90_Program
  ```

- ```
  RECURSIVE  FUNCTION  factorial (n)  RESULT (res)
    INTEGER  n, res
    res = 1
    IF ( n > 1 )  res = n * factorial(n-1)
  END FUNCTION  factorial
  ```

- ```
  SUBROUTINE  swap_structures (left, right)
    TYPE(structure) ::  left, right, temp
    temp = left ;  left = right ;  right = temp ;
  END SUBROUTINE  swap_structures
  ```

- ```
  MODULE  INTERVAL_ARITHMETIC
   ! type and operator definitions, other declarations
  CONTAINS
   ! module procedures
  END MODULE  INTERVAL_ARITHMETIC
  ```

- ```
  TYPE  INTERVAL
    REAL ::  inf, sup
  END TYPE  INTERVAL
  ```

- ```
  INTERFACE
    RECURSIVE  FUNCTION  factorial (n)  RESULT (res)
      INTEGER  n, res
    END FUNCTION  factorial
  END INTERFACE
  ```

Control Structures

- indexed_loop:
```
DO i = 1, 19, 2
  x(i+1) = x(i) - x(i-1)
END DO indexed_loop
```

- infinite_loop:
```
DO          ! is this an infinite loop?
  x = COS(x)
  IF ( x > 0.5 )  EXIT infinite_loop
END DO infinite_loop
```

- general_loop:
```
DO m = 6, 0, -1
  IF (MOD(m,3) == 0) CYCLE general_loop
  WRITE (*,*)  m   ! output: 5, 4, 2, 1
END DO general_loop
```

- while_loop:
```
DO WHILE ( x > 0.001 )  ! redundant !
  x = SIN(x)            ! feature   !
  WRITE (*,*)  x
END DO while_loop
```

- pseudo_while_loop:
```
pseudo_while_loop:   &    ! functionally equivalent !
&              DO         ! to while_loop above     !
  IF ( x <= 0.001 )  EXIT
  x = SIN(x)
  WRITE (*,*)  x
END DO pseudo_while_loop
```

```
• masked_array_assignment:                        &
  &          WHERE ( matrix1 > 0 )
               matrix2 = LOG(matrix1)
             ELSEWHERE
               matrix2 = -1000
             END WHERE masked_array_assignment
```

```
• choice:    SELECT ( number )
               CASE (  :0 )  choice
                 number = 0
               CASE ( 1:5 )  choice
                ! do not change 1, 2, 3, 4, 5
               CASE ( 7, 9 ) choice
                 number = number - 1
               CASE DEFAULT  choice ! CASE ( 6, 8, 10: )
                 number = 2 * number
             END SELECT choice
```

```
• block_if: IF ( number<=0 ) THEN  ! functionally equiv.
                 number = 0          ! to construct above
               ELSEIF ( 1<=number .AND. number<=5 ) THEN
                ! do not change 1, 2, 3, 4, 5
               ELSEIF ( number==7 .OR. number==9 ) THEN
                 number = number - 1
               ELSE
                 number = 2 * number
             END IF block_if
```

Intrinsic Types

- as in FORTRAN 77: `INTEGER`, `REAL`, `COMPLEX`, `LOGICAL`, `CHARACTER`
- all intrinsic types are now parametrized by a `KIND` type parameter to select a specific representation method (e. g. single, double, extended)

Examples:

```
INTEGER, PARAMETER:: DOUBLE=KIND(0D0), bit=1, Deutsch=49

INTEGER(KIND=2)                    ! may be 2 byte INTEGER
INTEGER(SELECTED_INT_KIND(9))      ! min. 9 decimal digits
REAL(KIND=KIND(0.0))               ! REAL
REAL(KIND=DOUBLE)                  ! DOUBLE PRECISION
REAL(SELECTED_REAL_KIND(24,300))   ! min. 24 dec. digits,
                                   ! exponent range +/-300
                                   ! same for COMPLEX:
COMPLEX(KIND(0.0))                   ! COMPLEX
COMPLEX(DOUBLE)                      ! DOUBLE COMPLEX
COMPLEX(SELECTED_REAL_KIND(24,300))  ! same as for REAL
LOGICAL(KIND(.FALSE.))             ! default LOGICAL
LOGICAL(bit)                       ! could be BIT type
CHARACTER(KIND("A"))               ! default character set
CHARACTER(Deutsch)                 ! German character set

   -12345_2                        ! INTEGER(2) constant
   1.23456789012E+3_DOUBLE         ! REAL(DOUBLE) constant
   (-1.333333333333_DOUBLE,4)      ! COMPLEX(DOUBLE) constant
   .TRUE._bit                      ! LOGICAL(bit) constant
   Deutsch_"eine längst fällige, höchst begrüßenswerte Änderung"
                                   ! CHARACTER(Deutsch) constant
```

Derived Types

- user-defined data structures with components of arbitrary types
- recursive type definition allowed if component is a pointer
- structure constructor uses derived type name as "constructor function"
- component selection introduced by "%" ("." in Pascal-like languages)
- assignment predefined for derived types, but may be redefined

Examples:

```
TYPE  INTERVAL  ! derived types cannot be parametrized
  REAL  INF, SUP
END TYPE  INTERVAL

TYPE  DOUBLE_INTERVAL    ! thus a new type is required
  REAL (DOUBLE)  INF, SUP
END TYPE  DOUBLE_INTERVAL

TYPE  list_of_names
  CHARACTER (LEN=30) ::  name
  TYPE (list_of_names), POINTER ::  next
END TYPE  list_of_names

TYPE (INTERVAL)  X, Y, Z
TYPE (DOUBLE_INTERVAL)  DI
TYPE (list_of_names)  list
  . . .
  ! structure constructor for INTERVAL object:
Z = INTERVAL( MIN(X%INF, Y%INF), MAX(X%SUP, Y%SUP) )
  . . .
list % next % name = 'Tom Smith'
```

Pointers

- `POINTER` is an attribute, not a special type
- pointers are strongly typed
- pointers are complete data object descriptors, not simply addresses
- pointer objects may have any type and any rank and shape
- `ASSOCIATED` tests whether a pointer is associated with a target
- one may `NULLIFY`, `ALLOCATE`, and `DEALLOCATE` pointers
- pointers are automatically dereferenced in most contexts
- context uniquely determines whether pointer or referenced object is used
- pointer assignment is denoted by `=>`
- any pointer target must have `TARGET` or `POINTER` attribute
- **Problem:** pointers are **initially undefined**, i. e. not NULL

Example:

```
TYPE  tree
  INTEGER ::  value
  TYPE (tree), POINTER ::  left, right
END TYPE  tree

TYPE (tree), POINTER ::  root, temp
 . . .
ALLOCATE ( root )
root%value = 3
NULLIFY ( root % left, root % right )
temp = root       ! ordinary assignment (copies object)
root%left => temp ! pointer assignment (copies pointer)
```

Arrays

- element type and rank (number of dimensions) always fixed
- assumed-shape arrays (dummy arguments with unspecified index bounds)
- automatic arrays (shape defined upon entry into a procedure)
- deferred-shape (dynamic) arrays: `ALLOCATABLE` or `POINTER` attribute
- one may `ALLOCATE` and `DEALLOCATE` deferred-shape arrays at any time
- deferred-shape arrays may be allocated with any shape (but fixed rank)
- `ALLOCATABLE` arrays must not be dummy arguments, function results, or structure components
- an array `POINTER` may be used wherever a pointer is allowed
- array constructors (one-dimensional): `(/ list_of_values /)`, e. g.
  `(/ 1, 3, 5, 7, 9 /)`, same as `(/ 2*i-1, i=1,5 /)`
- array sections (subarrays) via subscript triplets and vector subscripts
- subscript triplet: `(first:last:stride)`, e. g.
  `V(0:9:3)`, `V(9:0:-3)`, `V(6: :3)`, `M(2:4,2:4)`, `M(i, : )`
- vector subscript: `( index_vector )`, e. g.
  `V( (/0,3,6,9/) )`, `V( (/2,4,2/) )`, `M(2:4, (/2,4,2/))`
- zero-sized arrays allowed, e. g.
  `V(1:0)`, `M(2:4,4:2)`, `M(3,1:5:-2)`
- array assignment requires left-hand side array of appropriate shape
- **Problem: no automatic allocation** of left-hand side dynamic array during assignment

Examples:

- in Fortran 90, the multiplication of two non-square matrices is very cumbersome if the result matrix redefines one of the operands:

```
REAL, DIMENSION(:,:), ALLOCATABLE ::  A, B, T
  . . .
ALLOCATE ( A(m,n), B(n,k) )
READ (*,*)  A, B
  . . .
! now compute B = A * B !
ALLOCATE ( T(SIZE(A, 1), SIZE(B, 2)) )
T = MATMUL(A, B)
DEALLOCATE (B)
ALLOCATE ( B(SIZE(T, 1), SIZE(T, 2)) )
B = T
DEALLOCATE (T)
```

- in ACRITH–XSC, this is very simple (implicit memory management):

```
      DYNAMIC / REAL(:,:) /  A, B
        . . .
***   B is automatically freed and reallocated:
      B = A * B
```

- in Fortran 90, a function result (array) of unknown size must be a pointer:

```
FUNCTION  read_vector ( )
  REAL, DIMENSION(:), POINTER ::  read_vector
  INTEGER  dim
  READ (*,*)  dim
  ALLOCATE ( read_vector(dim) )
  READ (*,*)  read_vector
END FUNCTION  read_vector
```

```
PROGRAM  GAUSS   ! with allocatable arrays !

! Copyright (c) 1993 by Wolfgang V. Walter, Univ. Karlsruhe.
! May be copied and used as long as this notice is retained.

! Gauss elimination to solve a system of linear equations.
! The square system matrix A, the right-hand side vector b,
! and the solution vector x are ALLOCATABLE arrays.
! Array sections are used to access (parts of) rows of A
! and parts of the solution vector x.

  INTEGER              ::  i, j, n
  REAL                 ::  factor
  REAL, ALLOCATABLE ::  A(:,:), b(:), x(:)

  WRITE (*,*)  ' Please enter dimension n : '
  READ  (*,*)  n
  ALLOCATE ( A(n,n), b(n), x(n) )
  WRITE (*,*)  ' Please enter linear system row by row: '
  READ  (*,*)  ( A(i,:), b(i), i = 1, n )

  DO j = 1, n            ! reduction to upper triangular form
    DO i = j+1, n
      factor = A(i,j) / A(j,j)
      A(i,j+1: ) = A(i,j+1: ) - factor * A(j,j+1: )
      b(i) = b(i) - factor * b(j)
    END DO
  END DO

  x(n) = b(n) / A(n,n)  ! backsolving
  DO j = n-1, 1, -1
    x(j) = (b(j)-DOT_PRODUCT(A(j,j+1: ), x(j+1: ))) / A(j,j)
  END DO
  WRITE (*,*)  ' Solution vector x : ', x
END PROGRAM  GAUSS
```

```
PROGRAM  GAUSS   ! with array pointers as aliases !

! Copyright (c) 1993 by Wolfgang V. Walter, Univ. Karlsruhe.
! May be copied and used as long as this notice is retained.

! Gauss elimination to solve a system of linear equations.
! The system is stored in an (n x n+1) ALLOCATABLE array LS
! whose last column is the right-hand side of the system.
! The square system matrix A and the right-hand side vector
! b are pointers to array sections of LS (aliases).

  INTEGER                                      ::  i, j, n
  REAL                                         ::  factor
  REAL, DIMENSION(:,:), ALLOCATABLE, TARGET ::  LS
  REAL, DIMENSION(:)   , ALLOCATABLE          ::  x
  REAL, DIMENSION(:,:), POINTER               ::  A
  REAL, DIMENSION(:)   , POINTER               ::  b

  WRITE (*,*)  ' Please enter dimension n : '
  READ  (*,*)  n
  ALLOCATE ( LS(n,n+1), x(n) )
  WRITE (*,*)  ' Please enter linear system row by row: '
  READ  (*,*)  ( LS(i, : ), i = 1, n )  ! read full rows
                                        ! (A and b simult.)
  A => LS(1:n,1:n)      ! alias for system matrix
  b => LS(1:n,n+1)      ! alias for right-hand side

  DO j = 1, n           ! reduction to upper triangular form
    DO i = j+1, n
      factor = A(i,j) / A(j,j)
      LS(i,j+1: ) = LS(i,j+1: ) - factor * LS(j,j+1: )
    END DO              ! treat complete equation (A and b)
  END DO

  x(n) = b(n) / A(n,n)  ! backsolving
  DO j = n-1, 1, -1
    x(j) = (b(j)-DOT_PRODUCT(A(j,j+1: ), x(j+1: ))) / A(j,j)
  END DO
  WRITE (*,*)  ' Solution vector x : ', x
END PROGRAM  GAUSS
```

Modules and Interface Blocks

- `MODULE` encapsulates abstract data types (data type plus operations)
- `MODULE` encapsulates shared data (replaces `COMMON` blocks)
- `MODULE` encapsulates procedures and interfaces ("library")
- `PUBLIC` and `PRIVATE` control exportation to other modules (visibility)
- `USE` controls importation from other modules
- module procedures allowed at end of module, separated by `CONTAINS`
- local (nested) procedures allowed in external and module procedures and in main programs
- local procedures at end of host procedure, separated by `CONTAINS`
- nesting of procedures one level deep only (!)
- `INTERFACE` blocks define specific and generic interfaces
- a specific interface defines the interface of an `EXTERNAL` procedure
- a generic interface defines a generic procedure name, an `OPERATOR`, or an `ASSIGNMENT`
- overloading of intrinsic operators and procedures possible for new types
- user may define new operator names (no digits allowed), e. g.
  `.MUL.`, `.my_operator.`, `.Inverse.`
- unlimited overloading as long as interface is unique

```
MODULE  REAL_INTERVALS
! Copyright (c) 1992 by Wolfgang V. Walter, Univ. Karlsruhe.   !
! May be copied and used as long as this notice is retained.   !

! This module defines interval arithmetic with 2 ulp accuracy. !
! For the module to function properly, it is crucial that the  !
! real intrinsic operators  +, -, *, /  be accurate to 1 ulp.  !

  USE ERRORS    ! Import error handling routines.  A subroutine !
                ! ERROR(CHARACTER*(*)) is assumed to exist.     !

  PRIVATE  ! Everything that is not declared PUBLIC is PRIVATE !

  PUBLIC:: INTERVAL, IVAL, INF, SUP,                           &
 &         OPERATOR(+), OPERATOR(-), OPERATOR(*), OPERATOR(/), &
 &         OPERATOR(.IS.), OPERATOR(.CH.), OPERATOR(.SB.),     &
 &         OPERATOR(.SP.), OPERATOR(.DJ.), OPERATOR(.IN.)

  TYPE INTERVAL
    PRIVATE      ! (internal structure) components are PRIVATE !
    REAL INF, SUP
  END TYPE INTERVAL

  INTERFACE IVAL
    MODULE PROCEDURE IVAL
  END INTERFACE

  INTERFACE INF
    MODULE PROCEDURE INF
  END INTERFACE

  INTERFACE SUP
    MODULE PROCEDURE SUP
  END INTERFACE

  INTERFACE OPERATOR  ( + )
    MODULE PROCEDURE POS, ADD
  END INTERFACE
```

```
  INTERFACE OPERATOR  ( - )
    MODULE PROCEDURE NEG, SUB
  END INTERFACE

  INTERFACE OPERATOR  ( * )
    MODULE PROCEDURE MUL
  END INTERFACE

  INTERFACE OPERATOR  ( / )
    MODULE PROCEDURE DIV
  END INTERFACE

  INTERFACE OPERATOR  ( .IS. )
    MODULE PROCEDURE INTERSECTION
  END INTERFACE

  INTERFACE OPERATOR  ( .CH. )
    MODULE PROCEDURE CONVEX_HULL
  END INTERFACE

  INTERFACE OPERATOR  ( .SB. )
    MODULE PROCEDURE SUBSET
  END INTERFACE

  INTERFACE OPERATOR  ( .SP. )
    MODULE PROCEDURE SUPERSET
  END INTERFACE

  INTERFACE OPERATOR  ( .DJ. )
    MODULE PROCEDURE DISJOINT
  END INTERFACE

  INTERFACE OPERATOR  ( .IN. )
    MODULE PROCEDURE IN
  END INTERFACE


CONTAINS
```

```
TYPE (INTERVAL) FUNCTION IVAL (L, R)
  REAL           ::  L
  REAL, OPTIONAL ::  R
  IVAL%INF = L
  IF ( PRESENT(R) ) THEN
    IVAL%SUP = R
  ELSE
    IVAL%SUP = L                          ! point interval
  END IF
END FUNCTION IVAL

REAL FUNCTION INF (INTVAL)
  TYPE (INTERVAL) INTVAL
  INF = INTVAL%INF
END FUNCTION INF

REAL FUNCTION SUP (INTVAL)
  TYPE (INTERVAL) INTVAL
  SUP = INTVAL%SUP
END FUNCTION SUP

FUNCTION POS (INTVAL)                     ! unary +
  TYPE (INTERVAL) INTVAL, POS
  POS = INTVAL
END FUNCTION POS

FUNCTION NEG (INTVAL)                     ! unary -
  TYPE (INTERVAL) INTVAL, NEG
  NEG%INF = -INTVAL%SUP
  NEG%SUP = -INTVAL%INF
END FUNCTION NEG

FUNCTION ADD (L, R)
  TYPE (INTERVAL) L, R, ADD
  ADD%INF = NEAREST(L%INF+R%INF, -1.0)  ! approximation - 1ulp
  ADD%SUP = NEAREST(L%SUP+R%SUP, +1.0)  ! approximation + 1ulp
END FUNCTION ADD
```

```
FUNCTION SUB (L, R)
  TYPE (INTERVAL) L, R, SUB
  SUB%INF = NEAREST(L%INF-R%SUP, -1.0)  ! approximation - 1ulp
  SUB%SUP = NEAREST(L%SUP-R%INF, +1.0)  ! approximation + 1ulp
END FUNCTION SUB

FUNCTION MUL (L, R)
  TYPE (INTERVAL) L, R, MUL
  REAL  II, IS, SI, SS
  II = L%INF * R%INF ;   IS = L%INF * R%SUP
  SI = L%SUP * R%INF ;   SS = L%SUP * R%SUP
  MUL%INF = NEAREST( MIN(II,IS,SI,SS), -1.0 )
  MUL%SUP = NEAREST( MAX(II,IS,SI,SS), +1.0 )
END FUNCTION MUL

FUNCTION DIV (L, R)
  TYPE (INTERVAL) L, R, DIV
  REAL  II, IS, SI, SS
  IF ( 0.0 .IN. R )                                          &
 &  CALL ERROR('Division by interval containing zero')
  II = L%INF / R%INF ;   IS = L%INF / R%SUP
  SI = L%SUP / R%INF ;   SS = L%SUP / R%SUP
  DIV%INF = NEAREST( MIN(II,IS,SI,SS), -1.0 )
  DIV%SUP = NEAREST( MAX(II,IS,SI,SS), +1.0 )
END FUNCTION DIV

FUNCTION INTERSECTION (L, R)
  TYPE (INTERVAL) L, R, INTERSECTION
  IF ( .NOT. (L .DJ. R) ) THEN
    INTERSECTION%INF = MAX(L%INF, R%INF)
    INTERSECTION%SUP = MIN(L%SUP, R%SUP)
  ELSE
    CALL ERROR('Intersection of disjoint intervals')
  END IF
END FUNCTION INTERSECTION
```

```
  FUNCTION CONVEX_HULL (L, R)
    TYPE (INTERVAL) L, R, CONVEX_HULL
    CONVEX_HULL%INF = MIN(L%INF, R%INF)
    CONVEX_HULL%SUP = MAX(L%SUP, R%SUP)
  END FUNCTION CONVEX_HULL

  LOGICAL FUNCTION SUBSET (L, R)
    TYPE (INTERVAL) L, R
    SUBSET = L%INF >= R%INF .AND. L%SUP <= R%SUP
  END FUNCTION SUBSET

  LOGICAL FUNCTION SUPERSET (L, R)
    TYPE (INTERVAL) L, R
    SUPERSET = L%INF <= R%INF .AND. L%SUP >= R%SUP
  END FUNCTION SUPERSET

  LOGICAL FUNCTION DISJOINT (L, R)
    TYPE (INTERVAL) L, R
    DISJOINT = L%SUP < R%INF .OR. L%INF > R%SUP
  END FUNCTION DISJOINT

  LOGICAL FUNCTION IN (POINT, INTVAL)
    REAL POINT
    TYPE (INTERVAL) INTVAL
    IN = POINT >= INTVAL%INF .AND. POINT <= INTVAL%SUP
  END FUNCTION IN

END MODULE  REAL_INTERVALS
```

Other Features

- RECURSIVE functions and subroutines
- RESULT variable for function result
- INTENT(IN), INTENT(OUT), INTENT(INOUT) arguments
- OPTIONAL arguments
- PRESENT inquiry function to test for presence of optional arguments
- argument keywords (dummy argument names for argument identification)
- non-advancing I/O to read/write partial records (ADVANCE='NO')
- NAMELIST I/O to read/write data with variable names
- most FORTRAN 77 intrinsic functions are now elemental, i. e. applicable to arrays (no loops required)
- many new intrinsic functions for numeric manipulation and inquiry, array construction, reduction, reshaping, manipulation, inquiry, ...
- bit intrinsics for INTEGER (MIL-STD-1753)
- intrinsic subroutines for random numbers, bit copying, date and time
- binary, octal, and hexadecimal constants and I/O for INTEGER
- INCLUDE line to include source text (not a statement)
- language evolution: obsolescent features (which were already redundant in FORTRAN 77) are likely candidates for deletion from future standards, namely: arithmetic IF, non-integer DO loop control, shared DO loop termination, DO termination on statements other than END DO or CONTINUE, branching to END IF from outside the IF block, alternate return, PAUSE, ASSIGN, assigned GO TO and assigned FORMAT specifiers, H edit descriptor (Hollerith constants)

Summary

- Fortran 90 is probably the most powerful, versatile, and efficient internationally standardized programming language for numerical programming
- new language features of Fortran 90 greatly improve readability, maintainability, and portability of Fortran programs
- however, computational results are still often unreliable and incompatible when using various compilers, code optimization and vectorization strategies, and hardware platforms
- Fortran 90 standard fails to specify the mathematical behavior and accuracy of the arithmetic operators and mathematical functions
- Fortran 90 still does not provide any access to the IEEE operations with directed roundings toward $-\infty$ and $+\infty$, e. g. via the operators:

  `+<`, `+>`, `-<`, `->`, `*<`, `*>`, `/<`, `/>`
- Fortran 90 still lacks any kind of exception handling
- many full implementations of Fortran 90 to be expected in 1993 and 1994
- High Performance Fortran Forum (HPFF): concerted effort of parallel computer vendors and parallelization experts since Jan. 1992:
  - High Performance Fortran (HPF) is a Fortran 90 extension for (massively) parallel programming
  - intended for a variety of architectures: MIMD and SIMD, distributed and shared memory, vector processors, ...
  - uses directives only (Fortran 90 comments beginning with "`!HPF$`")
  - related to Vienna Fortran and Fortran-D
  - final document published in May 1993

Fortran 90 Literature

1. ISO: *Information technology – Programming languages – Fortran.* International standard ISO/IEC 1539 : 1991 (E), ISO Publ. Dept., Case postale 56, CH-1211 Genève 20, Switzerland, 1991.
2. ANSI: *American National Standard Programming Language Extended Fortran.* ANSI X3.198-1992, 1992.
3. DIN: *Informationstechnik Programmiersprachen: Fortran.* DIN EN 21539 : 1992, Beuth-Verlag, Berlin, 1993.
4. ISO: *Varying Length Character Strings in Fortran.* Draft int. std. ISO/IEC CD 1539-1 (collateral std. to ISO/IEC 1539 : 1991 (E)), 1992.
5. Adams, J. C.; Brainerd, W. S.; Martin, J. T.; Smith, B. T.; Wagener, J. L.: *Fortran 90 Handbook; Complete ANSI/ISO Reference.* Intertext Publ., McGraw-Hill, New York, 1992.
6. Brainerd, W. S.; Goldberg, Ch. H.; Adams, J. C.: *Programmer's Guide to Fortran 90.* Intertext Publ., McGraw-Hill, New York, 1990.
7. Metcalf, M.; Reid, J.: *Fortran 90 Explained.* Oxford Univ. Press, 1990.
8. Morgan, J. S.; Schonfelder, J. L.: *An Introduction to Programming in Fortran 90.* Blackwells, Oxford, to be published in 1993.
9. Aberti: *Fortran 90; Initiation à partir du FORTRAN 77.* S. I. editions, 2, rue du Fossan, F-06500 Menton, France, 1992.
10. Gehrke, W.: *Fortran 90 Referenz-Handbuch.* Hanser, 1991.
11. Heisterkamp, M.: *Fortran 90; Eine informelle Einführung.* B.I. Wissenschaftsverlag, Mannheim, 1991.
12. RRZN: *Fortran 90 Sprachumfang; Ein Nachschlagewerk.* Regionales Rechenzentrum für Niedersachsen, SPR.F90 1, Univ. Hannover, 1992.
13. Walter, W. V.: *Fortran 90: Was bringt der neue Fortran–Standard für das numerische Programmieren?* In: Jahrbuch Überblicke Mathematik 1991, 151–175, Vieweg, Braunschweig, 1991.